INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Sather 2: A Language Design for Safe, High-Performance Computing

Benedict Gomes, Welf Löwe,
Jürgen W. Quittek, Boris Weissman

TR-97-035
December 1997

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 an object model, 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.

Based on this model we define extensions for guarded suspension and data parallel programming. The model and the extensions are defined as a language proposal for a new version of Sather, Sather 2. The proposal is based on Sather 1.1, but replaces the parallel extensions of this version.

1 Introduction

With the advent of commodity multi-processors and cheap high-performance networks (Myrinet, ATM), parallel hardware platforms are now more widely available than at any time in the past. 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. However, most concurrent programming models are inadequate for these systems; it is currently extremely hard to realize the promise of parallel hardware on problems of significant size, even with highly skilled programmers.

Traditional programming models have either focussed on writing safe and correct programs (Actors, Eiffel etc.) or on writing efficient, high-performance programs (threads in C, C++ dialects). In reality, of course, both **safety** and **efficiency** are vitally important to realizing the potential of the parallel hardware available today. To safety and efficiency, we also add **ease of expression** as a primary consideration: in order to minimize the errors that occur in realizing a parallel application, the programming model must permit a natural expression of the application domain. These goals are frequently at odds, and previous approaches, such as Emerald [9], have attempted to resolve these conflicts in various ways.

This language proposal is another step in the same direction. It proposes a new parallel extension for the Sather language, replacing the parallel extension of Sather 1.1 [44]. It is intended to become part of the next version (2.0) of Sather called Sather 2. The base for the extension proposed in this document is serial Sather 1.1, although it might be changed for Sather 2. These possible changes are expected to be minor and to have no significant effect on the parallel extension.

The proposed new object model, though appearing to be fundamental, does not interfere with the object model for serial Sather, because the new properties affect concurrent programs only. However, some questions concerning the integration of this parallel extension within Sather 2 remain open, e.g. whether the serial Sather library will get complete method and and argument annotations corresponding to the visitor/mutator concept introduced in section 2.3. This might be necessary to use a common library for serial and concurrent programs.

The rest of this proposal may be divided into the following sections

- The remainder of this introduction is devoted to our basic approach and an overview of the other language models that most influenced our design.
- Section 2 describes the object model, starting with a simple, safe, but inefficient model and gradually increasing the sophistication of the model. Subtyping rules for this model are given in section 3.
- Section 4 deals with synchronous and asynchronous messages between objects. They are the basic way of creating concurrency.
- Section 5 introduces a system for designing complex synchronization constructs within the language.

- While the basic object model is appropriate for dealing with complex object structures, it is not appropriate for dealing with arrays and other flat structures. Section 6 describes the data parallel features which have been integrated into the basic language design.
- Appendix A explains the use of the new object model by giving a code example of a shield class implementing a bag.

1.1 Approach

Approaches to parallel programming may be divided into two broad categories. In implicit models, the parallelism in the application is discovered and exploited by the compiler and run-time system. In explicit models the parallelism is specified by the programmer. Between these two extremes, there are a variety of systems in which the programmer uses annotations to aid the compiler and run-time. We choose the explicit approach, and furthermore we choose to avoid reliance on heroic compiler analysis since achieving automatic parallelization is extremely hard and the achieved performance is frequently fragile.

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.

1.2 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 [44][14], Java [8][24], Ada [6], and C++ extended with thread and synchronization libraries such as POSIX threads [36] 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 state-

ments) 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 [13].

Safety at the Object Level

Many Actor languages [3] 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 [22]. 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 [26] 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 [25]. 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 [9] 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 [16]. 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 [15] 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 [5] 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)[1]. 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 proposed in this document 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.

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 [38], RSO [39], processor consistency [1], numerous flavors of release consistency [12], and other models supported only by the individual hardware vendors (DEC ALPHA [37], PowerPC [31], PentiumPro [18], etc.). As do many others [2][1][12][13], 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 [23] is the most natural candidate for such a top level programming model. Sequential consistency is central to the notion of object safety in the proposed language. The great

1. Protection against dynamic aliasing using "opaque" balloon types is mentioned in [5], but the details presented are only concerned with immutable objects.

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 [13]. Unlike other approaches that concentrate on detection of deviations from sequentially consistent executions by developing program analysis tools [2], 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 2.1.

2 The Object Model

Since the safety of the model is a hard constraint, we start by considering a design that provides safety and introduce features for expressiveness and greater efficiency while maintaining safety. The following sections introduce successively more relaxed models. Italicized terms are defined in greater detail in their respective sections.

- As a starting point, Model A in section 2.2 introduces a very simple model in which all operations on objects are serialized, thus ensuring safety at a high cost in performance (from the performance standpoint, this is similar to pure actors.)
- The *visitor/mutator* model (Model B) described in section 2.3 permits intra-object concurrency by allowing multiple reader methods to execute concurrently.
- The object aggregation Model C described in section 2.4 introduces coarser grained protection by requiring *shield* objects to furnish the protection needed for *interior* objects within a *domain* that the shield class controls. Interior objects cannot move between the domains of different.
- Model D in section 2.5 relaxes the requirement that interior classes be fully contained within a domain by permitting temporary sharing when such sharing cannot result in the object being *captured* by another domain.
- Model E in section 2.6 permits permanent transfer of interior objects to a different domains, provided that the objects have been *freed* from their original domain.
- Model F in section 2.7 describes the dynamic delegation of protection and synchronization that enables shield object aggregates.

2.1 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 2.4.2. 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 [2]. 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 2.2 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 [2][1][12][13], 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. [23]

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 [13].

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 [13]. 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.

Model A

2.2 The Simple Model

We start with a simple, and trivially safe model. A simple way to achieve safety and sequential consistency is by requiring that all object accesses be serialized. In other words, a mutual exclusion lock is associated with each object, and all method accesses to the object must acquire the lock.

The degree of concurrency can be increased by sending asynchronous messages to other concurrent objects. An executing asynchronous message is also referred to as a *thread*.

2.3 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.

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

2.3.1 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 implicit 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 definition of an attribute a of type T:

```
attr a:T;
```

implies the reader and writer methods:

```
visit a:T;
mutate a(val:T);
```

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.

2.3.2 Coarsening Protection by Annotating Methods

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[2]. 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.

Methods without annotation do not synchronize. See the example of the equality test in sets, shown below.

2.3.3 Coarsening Protection by Annotating Arguments

It is sometimes necessary to claim exclusive access to several objects at once, so that they may be modified without any intervening calls i.e. to further restrict the allowed transitions of the object, by disallowing certain external calls for a while. This is achieved by jointly locking multiple objects, either for visiting or for mutating. We already annotate the locking of self by the keywords **visit** an0d **mutate**. Syntactically, joint locking is accomplished by applying visit or mutate annotations also to method parameters.

Our running example while describing the object model will consist of a set class. We start by considering the basic operations on the set class such as insert and streaming through the elements[3].

```
class CONC_SET{T} is
    mutate insert(e:T) is...
    mutate delete(e:T):T is...
    visit elt!:T is...
    visit contains(visit e:T):BOOL is...
    visit is_eq(visit s:SAME):BOOL is...
end;
```

In the above example the *contains* and *is_eq* methods may be called concurrently on the set. The equality test for two concurrent sets can be done by the following:

```
visit is_eq(visit arg_set:CONC_SET{E}):BOOL is
    if size /= set.size then return false; end;
    loop
        if ~contains(arg_set.elt!) then return false end;
    end;
    return true;
end;
```

In this example, both *self* and *set* are claimed atomically for visiting at method entry. Moreover, the lock on both objects is maintained for the entire method invocation and hence neither set can change until the equality test method terminates.

2.3.4 Problems

There are two kinds of problems with the object model so far.

2. Since the synchronization of object state accesses inside such methods may be trivially lifted by the compiler.
3. All objects at this stage are assumed to provide their own independent protection.

- *semantic*s: all methods ensure protection on their own - this may be too fine-grained as in the general case it is more convenient to think about a sequence of methods as executing atomically and any extra protection is both unnecessary and cumbersome.
- *performance*: a straightforward implementation of such a model results in a reader/writer synchronization per method invocation. We would like to guarantee high performance by the object model, not merely rely on the compiler optimizations such as inlining and static analysis to try to lift extra synchronization. In the absence (or failure) of these optimizations, our naive model will prohibit many instruction reorderings routinely performed by modern parallel platforms with weak memory consistency.

Model C

2.4 Shield Classes 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 CONC_SET example, for instance, if the method delete 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 2 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 2 *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, the proposed Sather 2 object model allows dynamic sharing of objects between different object groups while maintaining proper protection.

The approach is somewhat similar to the idea of balloon types [5], but shield and interior objects allow further mechanisms like temporary sharing and transfer of interior object as described in sections 2.5, 2.6, and 2.7.

An executing Sather 2 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.

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 1 presents a view of an executing Sather 2 program.

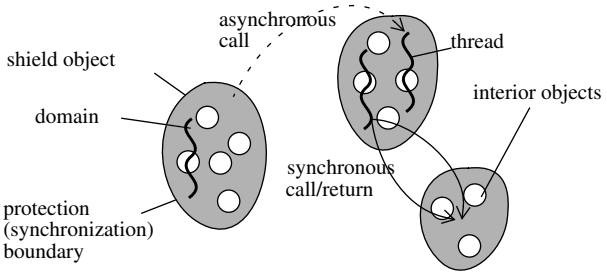

Figure 1: The Sather 2 Object Model

2.4.1 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.

We illustrate shield classes by using a concurrent set of interior objects.

```
shield class CONC_SET{INTERIOR_T} is
-- a concurrent set of interior elements
   private arr:ARRAY{INTERIOR_T};-- protected array - internal representation
   int size;                              -- set current size
   mutate create:SAME is
      res:SAME:= new;
      res.arr:= #ARRAY{INTERIOR_T}(res.asize);
```

```
        res.size = 0;
        return res;
    end;
end;
```

We will extend the set class as we concentrate on different features of the Sather 2 object model and provide enough information about Sather 2 to enable such step-by-step extension.

2.4.2 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.

2.4.3 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.

2.4.4 Problems

The main problem with this domain-based object model is that transferring interior objects between domains is not supported. Thus, all communication between threads must take place through shield classes.

Model D

2.5 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.

2.5.1 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.

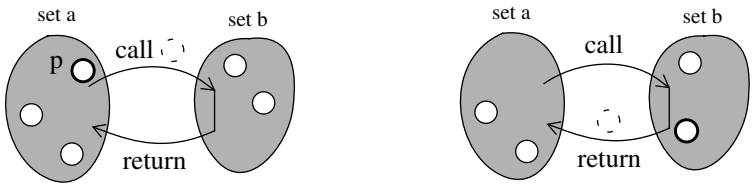

Figure 2: Interior objects and inter-domain calls

2.5.2 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 2. 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 3.

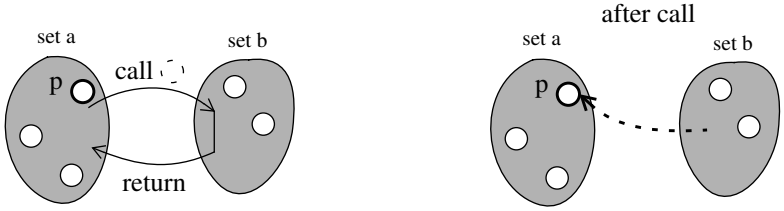

Figure 3: Reference capture and static inter-domain aliasing

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.

2.5.3 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 is_member(visit e:INTERIOR_T):BOOL
```

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

We permit 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. For instance, they validate the signature of *is_member* that we have previously used. The implementation of *is_member* is safe since the passed reference to a interior object cannot be captured. We now reexamine is_eq to make sure that its implementation is correct.

```
visit is_eq(visit arg_set:CONC_SET{PROT_E}):BOOL is
    if size /= arg_set.size then return false; end;
    -- now check for all element matches
    loop
        e;PROT_E:= elt!;
        if ~arg_set.contains(e) then return false end;
    end;
    return true;
end;
```

In this example, the only call that crosses domain boundary is *arg_set.contains(e)* The call is both legal and safe since it has a signature:

```
visit contains(visit e:T);
```

Since the called method is a visitor, it cannot capture a passed reference, and hence the passed element is shared only while the call is in progress.

We now have almost all information necessary to implement the rest of the concurrent set class. However, what is missing is the ability to transfer interior objects between domains.

2.5.4 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 2.4.3. Thus (A) is maintained. As described in Section 2.5.3, 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.

2.5.5 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.

2.5.6 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.

Model E

2.6 Transferring Interior Objects

It is sometimes necessary to transfer objects between interior objects between different object domains. For the reasons mentioned in Section 2.5.5, 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 se-

quential 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 2 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.

An 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 4 displays the object groups associated with several free objects.

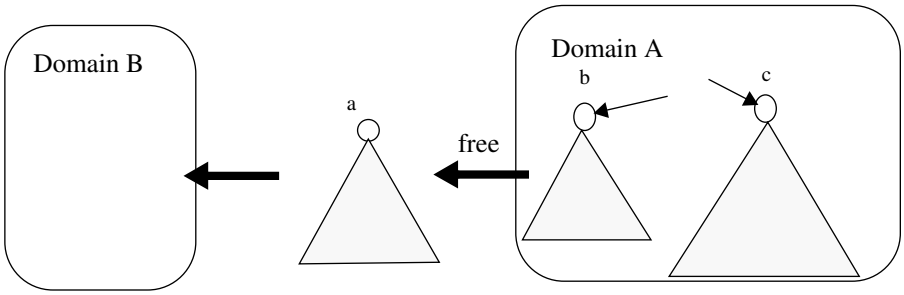

Figure 4: Free Object Groups

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 4.

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.

2.6.1 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.

2.6.2 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[4]. 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 visitor 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.

2.6.3 Shield Class Interface Restrictions

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

4. 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.

- 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.

2.6.4 Example

We may now extend the concurrent set class by adding the insert method:

```
mutate insert(free e:PROT_T) is
    if(size+1 > asize) then double_and copy; end;
    -- double the array and copy the original state
    arr[size] := e;
    size := size+1;
end;
```

Freeing an object is also useful for operations such as delete which remove all current references to the object. For instance, the following method releases an element that matches its argument:

```
mutate delete(e:PROT_T): free PROT_T is
    -- release and return a set element matching e, if found
    res:PROT_T;
    loop
        i::=0.upto!(size-1);
        if(e = arr[i]) is res = arr[i]; shift_left(arr, i, 1); break!; end;
    end;
    return free res;
end;

visit intersect(visit arg_set:CONC_SET{PROT_T}): CONC_SET{PROT_T}is
    -- computes an intersection of self with 'set' and returns it as a new set
    res: C_SET:= #;
    loop
        e:= arg_set.elt!; -- create a dynamic alias
        if contains(e) then res.insert(free e.copy) end;
    end;
    return res;
end;

visit union(visit arg_set:CONC_SET{PROT_T}): CONC_SET{PROT_T}is
    -- computes a union of self and 'set' and returns it as a new set
    res: C_SET:= #;
    loop res.insert(free elt!.copy); end;
    loop
        e:= arg_set.elt!;
        if ~res.contains(e) then res.insert(free e.copy) end;
    end;
    return res;
end;
```

The above code assumes the existence of a *copy* method in *PROT_T* that returns a free copy of self and has a signature:

```
    copy: free PROT_T;
```

2.7 Synchronization Aggregation

An important semantic benefit of the object aggregation scheme described in the proceeding section that groups of protected objects may be accessed within a concurrent method body, with the assurance of no external interference. This coarsening of locking is useful, for instance, when computing the sum of all the elements of a container.

For semantic reasons, it is desirable to support a similar coarsening of locking on concurrent objects. While annotating method arguments (as described in Section 2.3.3) allows us to conjunctively lock small numbers of concurrent objects, it is frequently useful to be able to lock a larger aggregation of objects. This permits operations to be performed on elements of the aggregate without external interference.

Aggregate protection may be achieved if the protection for the whole group is consolidated in some "protector" object which provides protection for the group as a whole. To achieve this transfer of protection, we define the *protector* of a concurrent object. All synchronized objects have an attribute protector bound to the protector object. A concurrent object is either protected by itself, in which case:

```
a.protector = a
```

or it is protected by some other object, b:

```
a.protector = b
```

Transfer of the protection of 'a' to 'b' must be signalled explicitly:

```
a.protector:= b;
```

Transfer of the protection of an object back to itself is achieved by:

```
a.protector:= a;
```

'a' is protected by 'b' implies that any visitor of 'a' is a visitor of 'b' and any mutator of 'a' is a mutator of 'b'.

Note that the synchronization aggregation of concurrent classes does not restrict their usage - they may still be freely used in other contexts. However, any access will result in the lock for the overall aggregate being claimed. Thus, if a set aggregates synchronization on all its elements, the elements may still be accessed outside the set. However, acquiring the lock on any of the elements will result in the aggregate lock being claimed.

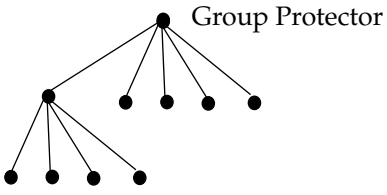

Figure 5: Tree of Aggregated Concurrent Objects

2.7.1 Performance Implications

The compiler and runtime may use this information to get rid of some fine-grained synchronization and consolidate all synchronization in the protector object. For instance, all elements of a concurrent array may be protected by the array object itself. Iterating over the elements of the array will acquire little or no synchronization overhead in addition to synchronizing on the array object. Concurrent array elements may be freely exported outside the array object boundary (unlike protected attributes). However, synchronization will remain coarse-grained even when such objects are used not as array elements.

2.7.2 Determining the Protection of Aggregates

Performing aggregation in a naive manner can require time linear in the size of the aggregate to determine the protector of an object. However, we have been investigating the use of modified union-find operations to reduce the average cost of aggregation:

A semi-dynamic problem occurs when there is a notion of state of objects that goes beyond only considering the attributes of that object. Assume the state of an object to be defined by its value attributes and the state of its reference attributes that are marked as *owned*. Additionally, it is required the each object knows its *master*. A master of an object o is defined as follows: (i) the master of o is o iff it has never been assigned to an attribute marked as owned or (ii) the master of o is the master of the object that contains the owned attribute o is assigned to[5]. Initially, each object is its master. Ownership may get coarser by assigning an object to an attribute marked as owned.

Simple Algorithm

The simple application of the Union-Find data structure would solves the problem: Assume, initially each object is a singleton set. The operation *find* applied on a object o identifies the master of o. Assigning an object o to an (marked owned) attribute of an object o' should have the effect that, according to the definition of master, $find(o)$ becomes $find(o')$ and $find(o')$ remains the same. However, this behavior cannot be guaranteed by the *union* operation. The reason is that $find(o')$ remains the unchanged iff $|find(o)| \leq |find(o')|$[6]. For a first algorithm, we drop this invariance. For various probability assumptions, the expected running time is linear for the above algorithm, cf. [4].

Advanced Algorithm

The next algorithm is an easy extension of the Union-Find data structure. First of all, we observe that the Union-Find data structure works also for on-line problems: Obviously, it is no problem to create new objects, i.e. to add new singleton sets, on-line. We add an operation *master* which, applied to an object, returns its master. As an invariance of the new data structure, each object o representing a set computes $master(o)$ in time $O(1)$, e.g. by

5. In Sather, e.g., synchronization is done by the master of an object. This master must therefore be accessed when an object is called.

6. Note that $find(o')$ denotes a set and $|find(o)|$ the cardinality of this set.

storing them in special attribute. Obviously, the invariance can be guaranteed for the initial singletons. For objects that do not represent a set, *master*(*o*) is defined by *master*(*find*(*o*)). It is also invariant that for any object *o*, *master*(*o*) is the master object of *o* (as defined above). Let *o* be an object to be assigned to an (marked owned) attribute of another object *o′*. In this case we find the master of *o′*, assign it to some auxiliary variable, say *new_master*, then execute *union*(*o*,*o′*), and finally redefine *master*(*o*) = *master*(*o′*) = *new_master*. Of course, this can be implemented in *O*(*find*). Obviously, it holds the following:

> For any sequence of assignments and requests to find a master object, for each object *o*, *master*(*o*) computes the master of *o*. Any sequence of n assignments and $m \geq n$ requests to find the master object requires time $O(n + m \times \alpha(m, n))$ with the above implementation where $\alpha(m, n)$ is the inverse of Ackermann's function.

3 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.

3.1 Abstract Interior Classes

Methods in abstract interior classes may be annotated as **visitable** or **mutable**. Methods marked visitable are considered visitor methods and methods marked mutable are considered mutator methods; protection for such methods must be propagated to all method calls, just as it is with concrete interior classes as described in Section 2.4.2. Abstract interior classes may only subtype from other abstract interior classes.

3.2 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.**

3.3 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.

3.4 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:

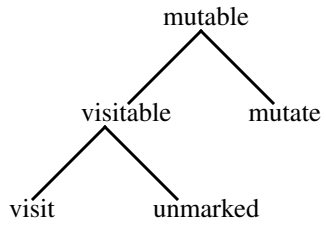

FIGURE 6. Extending Sather subtyping rules

3.5 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.

4 Threads and Concurrency

Similar to a “standard” object-oriented model, a Sather computation involves messages that trigger method execution. Sather messages are active - they can trigger an action at the destination object without cooperation of other threads, processes, or “bodies”.

4.1 Active Messages

There are two kinds of messages: synchronous and asynchronous:

- *Synchronous calls*. A thread executing a synchronous call relocates itself to the destination object by sending an *active* message to that object. Messages that do not satisfy a *proceed* criterion, i.e. that are waiting for synchronization, are queued until it is satisfied. On method termination, an active message containing the return value, if any, is sent back to the source object. Execution resumes immediately upon the arrival of a return message.
  The most common and trivial case of an active message carrying a synchronous call is a method call on a local object, which is executed with no run-time sysem overhead for the message.
- *Asynchronous calls*. Non-blocking calls are performed by creating a new thread and locating it to the destination object by sending an active message. The sending thread continues execution immediately. The start of a new thread signifies the creation of a new domain. Active messages carrying asynchronous calls can only be sent to shield (or immutable) objects. Such calls can have arguments of shield types, value types, and free interior types.

While generating synchronous calls is already covered by serial Sather, new language features are requoired to express asynchronous calls.

4.1.1 Fork Expressions

While generating synchronous calls is already covered by serial Sather, new language features are requoired to specify asynchronous calls. Asynchronous calls are performed by *fork* expressions with the following syntax:

fork_expr ⇒ fork *[bundle,] call_expression*

Fork expressions have the type FUTURE, if the method has no return value, or FUTURE{T}, if the method inside fork returns a value of type T. The following example creates a new thread to compute a sum of two immutable integers. Note that in Sather 1+2 is just syntactic sugar for 1.plus(2). The original thread then blocks until the result is available:

```
t:FUTURE{INT}:= fork (1+2);
sum:INT := t.get;  -- wait until the thread terminates
```

4.1.2 Futures

The interface of futures has only two methods: get and is_done. get is blocking - a thread that tries to perform a get operation on the future blocks until the corresponding thread is terminated. If the call has a return value, it is returned by get. is_done is non-blocking and returns a boolean informing the caller about the state of the future.

4.1.3 Thread Bundles

Thread bundle is an optional argument for a thread creation expression. It serves as a handle on a collection of threads and can be used, for instance, to wait until all threads in that collection terminate. The next section will provide more information on bundles, bundle operations, and thread scheduling.

4.1.4 Example

We now change the original sequential version of intersect() to exploit parallelism. The following code segment uses a very simple parallel algorithm: a separate thread is created for each element of the set to see if the element belongs to a set passed as an argument. If so, a copy of the element gets inserted into the resulting concurred set. Since threads are visiting self and argument ‘set’, the bulk of thread bodies can execute concurrently. The only mutually exclusive part is due to the insertion of new elements to the resulting set.

```
visit Intersect(visit set:CONC_SET{PROT_T}):CONC_SET{PROT_T} is
    -- a parallel version of intersect
    res:SAME:= #;
    bundle:$BUNDLE:= #BUNDLE; -- create a new default bundle
    loop
        i:= 0.upto!(size-1);
        future::= fork bundle, intersect_chunk(i, set, res);
    end;
    -- now simply wait until all done
    bundle.join;   -- wait until all threads in a bundle terminate
    return res;
end;

private visit intersect_chunk(i:INT, visitable set:SAME, mutable res;SAME) is
    -- compute an intersection of a range of elements in self starting with ‘start’
    -- of size ‘range’ with set ‘set’. Add the results to ‘res’
    loop
        e::= arr[i];
        if set.contains(e) then
            res.insert(e.copy);
        end;
    end;
end;
```

In this example, a new bundle of the default bundle type is created and is used to signal when all created threads terminate. The original thread blocks inside bundle.join until this happens.

4.2 Bundles and Scheduling

In this section, we discuss Sather *thread bundles*. Thread bundles, or simply *bundles,* are a collection of logically related threads with common properties. The most important shared property is that all groups in a bundle share the same scheduling policy.

Bundles fulfill several purposes:

- Similar to concurrent classes that provide aggregation of objects, bundles provide aggregation of activities.
- "Standard" thread synchronization operations can apply to entire bundles. For instance, a join method can be called on the bundle to wait until all threads in a bundle terminate.
- Different bundles support different scheduling policies. Similar to object hierarchies, activities can also be combined into hierarchies by associating them with appropriate bundles.

Although we haven't mentioned thread bundles much earlier, they have been around all along. All Sather threads, including the original thread that executes the main method belong to bundles. The ability to aggregate parallel activities will come very handy in the data parallel extension of Sather.

We now examine the fork expression in more detail:

```
f:FUTURE{T}:= fork([bundle:$BUNDLE,] method(args));
```

In the absence of an optional bundle argument, a new thread is added to the current thread's bundle. If the bundle argument is present, a thread is added to the specified bundle. When the thread terminates, it is automatically removed from its bundle.

4.2.1 Bundle Hierarchies

Bundles can form hierarchies. A newly created bundle is automatically added as a child to the bundle of the current thread. Fig shows a bundle hierarchy of an evolving Sather computation.

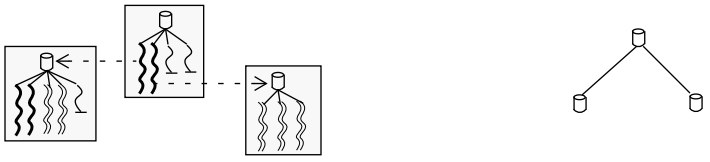

Figure 7: Creation of new bundles (left) and a resulting Bundle hierarchy (right)

A built-in expression current_bundle:$BUNDLE returns a bundle of the calling thread. Any bundle implementation must conform to the following interface:

```
abstract class $BUNDLE < $IS_EQ is
    parent:$BUNDLE:             -- returns the parent of a bundle
    child!:$BUNDLE;             -- iterates through child bundles
    num_threads:INT;            -- a “current” number of threads in the bundle
    num_blocked_threads:INT;    --”current” number of blocked threads
    join;                       -- called from outside: blocks until all threads terminate
    barrier_init;               -- “unit” the barrier; the following barrier will be done
                                -- for the # of threads at the moment of the call
    barrier;                    -- blocking barrier
    name:STR;                   -- “name” for debugging purposes
end;
```

$BUNDLE provides merely an interface all bundles must conform to. One can envision another abstract class for bundles that support priority-based scheduling:

```
$PRIORITY_BUNDLE < $BUNDLE is
    set_current_priority(p:INT); -- sets the priority of a calling thread
    get_current_priority:INT;    -- returns priority of a calling thread
end;
```

Since we have a two level system that captures the state of activities (threads and bundles), two kinds of scheduling decisions have to be made:

- scheduling of threads in a bundle
- scheduling of different bundles

4.2.2 Bundle Scheduling Policies

The current distribution of Active Threads (a compilation target for Sather) currently comes with the following bundles:

- threads scheduling policy: FIFO, LIFO, MCS and versions of these with lazy stack allocation.
- bundle scheduling policy: FIFO

4.2.3 Discussion

Superficially, Sather bundles are reminiscent of the Java thread groups [8]. Both entities serve to aggregate activities and can be combined into hierarchies. They also share one common goal - the aggregation of control over multiple activities. However, the rest of the goals are quite different: Java thread groups are mainly needed to ensure security in a distributed multi-user environment. In contrast, Sather bundles enable compositional development of parallel software and encapsulation of scheduling policies in program modules. For very fine-grained parallel applications, a careful use of bundles that support memory-conscious thread scheduling (to minimize the number of cache misses) leads to significant speedups and drastically reduced memory requirements. Both runtime and memory usage has been shown to improve by as much as an order of magnitude for some parallel platforms [47].

5 Guarded Methods and Properties

A main goal of our object model is sequential consistency on the object level. This implies synchronization of concurrent threads acting on objects. The concept of visitors and mutators provide a way to synchronize threads that can be used e.g. to protect a shared resource. But in general, a threaded concurrent programming language like Sather 2 should provide further means for synchronization than just reader/writer exclusion.

Condition variables, for example, cannot be implemented by the means of our object model without busy waiting which in the most cases leads to poor performance. Furthermore, a busy waiting implementation would mix synchronization code and general code. Readability would be poor and the condition variables could not be considered when statically checking for substitutability.

This chapter introduces guarded methods and properties extending the synchronization facilities of our object model. After stating the design goals in section 5.1, syntax and semantics of guarded methods and properties are introduced stepwise. Single guarded methods and basic properties are described in section 5.2. The basic concepts are extended in section 5.3 (extended properties) and section 5.4 (disjunctive guarded methods). Examples are given for all new language features. Section 5.5 discusses how they can be implemented efficiently.

5.1 Design Goals

The synchronization constructs introduced in this chapter meet the design goals of the whole language, namely safety, performance, and ease of expression. Additionally, they extend the object model in a natural way.

5.1.1 Natural Extension

In order to be consistent with our object model, we use the same synchronization points. A thread may synchronize with others only when it enters or leaves a method. This is supported by many synchronizing constructs of object-oriented concurrent programming languages like body methods of active objects (e.g. [7]), accept sets [19], enabled sets [46], and method guards (e.g. [10][29]). But only methods guards expose the synchronization constraint of a method at its signature and thereby allows save subtyping with respect to synchronization behavior like our object model does. Also syntactically, adding a guard to a method is closer to the visitor/mutator annotation of the object model.

5.1.2 Safety

There are three aspects of safety to be considered: keeping the safety of our object model, safe subtyping, and deadlock detection.

Keeping the Safety of the Object Model

In order to keep the safety of the object model, visitors/mutators and guards are combined conservatively. A thread is blocked, if at least one of these two mechanisms blocks it. Furthermore, calling a guarded method may not change the status of the caller being a reader or a writer, even if the call blocks.

Safe Subtyping

Safe subtyping is required to ensure substitutability of classes by their subclasses. Particularly for the design of robust class libraries, substitutability is essential. According to Meyer's concept of *design by contract* [32], the calling object (client) of a method has to ensure a precondition to hold when calling. In turn, the called object (server) guarantees a postcondition to hold when the method returns. Now, a class can safely be substituted by a subclass, if for all of its methods holds that the precondition implies the precondition of the corresponding method of the subclass and that the postcondition is implied by the postcondition of the corresponding method of the subclass.

In absence of concurrency it is a severe run-time error, if a precondition or postcondition does not hold. Systems capable of detecting such an error usually raise an exception when the contract is violated. For concurrent systems the situation is different. A precondition being false when a thread calls a method, might become true on account of another thread. Hence, it might make sense to block the thread until the precondition becomes true instead of raising an exception.

To solve this conflict, we partition the precondition of a method in two parts, the *immediate precondition* and the *blocking precondition*. While a violation of the immediate precondition is a run-time error, a violation of the blocking precondition blocks the calling thread until the condition becomes true. The blocking precondition is also referred to as *synchronization constraint*, *method guard*, or simply as *guard*.

There is no need to partition postconditions in a similar way, since they have to be guaranteed by the method when returning. But in order to increase readability and to support an efficient implementation we also partition postconditions (see below).

In Sather precondition and postcondition are part of a methods signature, though usually specified incompletely. By interpreting the method guard as part of the precondition, we can achieve safe substitutability also concerning synchronization constraints. This allows static checking blocking preconditions for subtypes, at least for basic cases.

Deadlock Detection

A safety goal of any multi-threaded language should be to avoid deadlocks by statically checkable restrictions. But usually, too strong restrictions though providing high safety are not desirable, because they reduce expressibility and might forbid the implementation of common design patterns. Hence, multi-threaded languages do not avoid deadlocks entirely.

In order to compensate this weakness, a further goal is to detect deadlocks at run-time as far as possible. Since run-time checks might have severe performance implications, we tried to reduce the synchronization points and to allow a scheme of signaling synchronization events between threads, that allows to detect a lot of deadlock situations with low effort. This supports development of deadlock free code, but does not give complete safety, because not all deadlocks can be detected.

5.1.3 Performance

An efficient implementation of visitor/mutator synchronization can make use of existing low level reader/writer locks or can be built easily upon other synchronization primitives as mutexes or semaphores. But for guarded methods the problem is much more complex. Several design issues being critical for performance have to be discussed including the questions when, how often, and by which thread a blocking guard is re-evaluated.

Synchronization Points

We support a low number of re-evaluations by the language definition. Guards are restricted to expressions on *properties*, special attributes (basic properties) or methods (extended properties) of shield objects. Changes to basic properties are only possible when a method returns. Hence, re-evaluation of guards depending on an object's basic properties is only required, if a method called on this object returns. These synchronization points, blocking on method entry and signaling on method return, are the same as in the object model, supporting a simple and efficient implementation for the combination of both. We keep this synchronization scheme also for extended properties, even though they might change before a modifying method returns.

Number of Re-Evaluations

The remaining number of necessary guard re-evaluation can be reduced by static analysis and dynamic tracking of guard dependencies. The analysis of dependencies is supported by demanding all changes to properties being stated in a methods postcondition. Like preconditions, postconditions are split in two parts. The first one, the checking postcondition, concerns everything but properties and may be incomplete. The second one, the signaling postcondition, describes the changes to properties made by this method completely. These changes are stated in the signaling postcondition only. Thus, the signaling postcondition is as well an assertion as it is executable code. This simplifies programming and the analysis of guard dependencies. It avoids contradictions between method body and postcondition and avoids undecidable situations which could occur if changes to properties would appear in the method body, e.g. in conditional branches.

Avoiding Context Switches

A further question concerning performance is which thread evaluates a blocked guard. In Java for example, only the blocked thread itself can re-evaluate its guard requiring context switching for each re-evaluation [8]. By restricting guards to expressions on properties, any

thread can re-evaluate a guard. In particular a thread signaling the need for a re-evaluation can re-evaluate all blocked guards without any context switch.

5.1.4 Ease of Expression

Parallel programming adds complexity to sequential programming. In order to reduce errors that occur in realizing parallel applications, the programming model must permit a natural expression of common design patterns in the application domain. We achieved a clear expression of synchronization constraints by separating synchronization code from method bodies and by making them a part of the signature. With the synchronization behavior of an object being reflected by its interface, it is possible to reason about synchronization without knowing the implementation. Code re-use is supported by avoiding most of the known inheritance anomalies caused by inherited synchronization code.

Object Properties

We clarified the specification of guards by adding *properties* to objects. Properties indicate states of the object with respect to its synchronization behavior. In our eyes, the most natural way of specifying synchronization behavior is to say 'execute this method when the object has certain properties'. Accordingly, we define guards as boolean expressions on properties. Extended properties allow sharing of complex (partial) blocking preconditions between methods by.

Separation of Synchronization Code

Partitioning preconditions and postconditions separates synchronization code from other actions on the object and allows designing it and reasoning about it separately from other code. Furthermore, the different semantics between immediate preconditions and checking postconditions on one side and blocking preconditions and signaling postconditions on the other side is reflected by the language. The former are assertions that may be omitted and that may describe the corresponding condition only partially. The are compiled to run-time checks that can be turned on or off by compiler switches. The latter always describe the corresponding condition completely and are compiled to code that is essential for execution. The separation also simplifies specifying synchronization constraints as part of an interface.

Avoiding Inheritance Anomalies

Another important issue concerning the ease of expression is code re-use. Inheritance anomalies, i.e. code inheritance and synchronization constraints conflict with each other, require re-definitions of inherited methods in order to maintain the integrity of synchronizing objects. In general, Sather reduces inheritance anomalies by separating interface inheritance (subtyping) and code inheritance (re-use). Since the blocking preconditions and the signaling postconditions are part of the interface, code inheritance is not necessarily affected by changes of synchronization constraints.

Furthermore, Matsuoka and Yonezawa [30] showed that method guards avoid a lot of inheritance anomalies. They characterized the remaining anomalies as *history-only sensitiveness*. These anomalies can occur when a subclass adds a method with a guard depending on the calling sequence of inherited methods before the actual call. These anomalies are hard to avoid, even the advanced solution Matsuoka and Yonezawa propose in [30] does not avoid them completely. Other anomalies in conjunction with guarded methods, which are claimed in this paper do not occur in Sather 2, because they violate the subtyping rules, in particular the implication of the bocking precondition in the subclass by the blocking precondition in the superclass.

5.2 Guarded Methods and Basic Properties

Guarded methods and properties are introduced stepwise. Similar to the object model, we start with a safe and simple but restricted design. This design already offers high performance. Its extension is mainly driven by gaining expressibility while keeping performance but slightly relaxing safety. This section introduces and discusses syntax and semantics of our starting point: single guarded methods and basic properties.

Syntax and Semantics

Any method of a shield class may be guarded by a *blocking precondition*. Like a (immediate) Sather precondition, a blocking precondition is part of the method's prologue and specified by a boolean expression. A thread may enter a guarded method only if the blocking precondition is true. At the same time, it must comply with the rules for visitors and mutators. Otherwise, the thread is blocked, until it may enter. Evaluating a blocking precondition and becoming a visitor or a mutator, resp. is one atomic operation.

The only identifiers visible inside a blocking precondition are *properties*. A property is a special boolean attribute of a shield class. It is read-only from outside the class and its declaration is preceded by the keyword property.

property_definition ⇒ property *property_identifier_list* : BOOL
blocking_precondition ⇒ blocking_pre *property_expression*

Besides the blocking precondition, a method of a shield class may have a *signaling postcondition*. Like the blocking precondition, it is specified by a boolean expression, but the expression is restricted. It exclusively consists of comparisons for equality between properties and an initial expression on properties or a boolean literal. These comparisons are combined by the boolean and-operation.

signaling_postcondition ⇒ signaling_post *signaling_post_expression*
signaling_post_expression ⇒ *property_comparison*
| *signaling_post_expression and property_comparison*
property_comparison ⇒ *property_identifier* = *initial_property_expression*
initial_property_expression ⇒ initial (*property_expression*) | true | false

Though syntactically being a boolean expression, the signaling postcondition mutates the objects state by modifying the objects properties such that itself becomes true. It is a fatal

error, if this is not possible. The required operations on the properties are executed atomically when the method returns. If a signaling postcondition is present, the corresponding method is a mutator.

Signaling postconditions have further semantics. They trigger the re-evaluation of blocking preconditions for threads being blocked. Since the signaling postconditions describe the state transitions of properties exactly, it can be decided statically, which blocking precondition has to be re-evaluated after a method with a blocking postcondition has been executed. These re-evaluations are executed after the guarded method is left and before any other synchronized method of the object is entered by any thread. If a blocked thread's blocking precondition is implied by a signaling postcondition, it even does not have to be re-evaluated. By this means the number of re-evaluations of blocking conditions can be minimized.

A blocking precondition and a signaling postcondition are part of the method's signature. The inheritance rule of co/contravariance applies also to them, i.e., the blocking precondition of the supertype must imply the blocking precondition of the subtype and the signaling postcondition specified for the subtype must imply the signaling postcondition specified for the supertype. Consequently, properties are part of the class interface.

Discussion

In general, guarded methods extend the reader/writer protection of objects to a more general restriction of the legal calling sequences on objects. Legal sequences are ensured by blocking illegal calls. A call can be interpreted as a transition of the object's synchronization state with basic properties being the state variables. If no method contains calls to further blocking objects, then the synchronization behavior of the object can be described as a deterministic finite state machine (DFSM) and all legal calling sequences are regular expressions over the interface. In this notion, subtyping means extending the DFSM such that the DFSM of the superclass is contained in the subclass's DFSM. Subtyping of such classes can be checked efficiently.

As an example, a simple parametrized buffer class has been chosen, capable of keeping a single object of type T. The abstract class $BUFFER{T} defines the interface including the complete specification of the synchronization behavior. The concrete shield class BUFFER{T} gives an implementation of the interface.

```
abstract shield class $BUFFER{T} is
    property full:BOOL;
    create:SAME;
    mutator put(item:T) blocking_pre ~full signaling_post full = true;
    mutator get:T blocking_pre full signaling_post full = false;
end;

shield class BUFFER{T} < $BUFFER{T} is
    attr buffer:T;
    property full:BOOL;
    create:SAME is
        return new.init; end;
    private mutator init:SAME signaling_post full = true is
        return self; end;
    mutator put(item:T) blocking_pre ~full signaling_post full = true is
```

```
        buffer:= item; end;
    mutator get:T blocking_pre full signaling_post full = false is
        return buffer; end;
end;
```

Here is a very similar class, a future buffer (set once, read multiple times) having the same interface as the buffer above except for the postcondition of method get.

```
shield class FUTURE_BUFFER{T} is
    property full:BOOL;
    create:SAME;
    mutator put(item:T) blocking_pre ~full signaling_post full = true;
    visitor get:T blocking_pre full;
end;

shield class FUTURE_BUFFER{T} is
    attr buffer:T;
    property full:BOOL;
    create:SAME is return new.init; end;
    mutator init:SAME signaling_post full = false is return self; end;
    mutator put(item:T) blocking_pre ~full signaling_post full = true is buffer := item; end;
    visitor get:T blocking_pre full is return buffer; end;
end;
```

Their synchronization behavior of these classes is entirely visible at their interfaces. Without specifying it in the interface, $BUFFER and $FUTURE_BUFFER would be identical, even though they are not substitutable for each other in both ways. An Analysis of the synchronization behavior shows that a $FUTURE_BUFFER can safely be substituted by a $BUFFER, but not vice versa. Correspondingly, an application of our subtyping rules considering synchronization behavior shows that $BUFFER is a subtype of $FUTURE_BUFFER, but $FUTURE_BUFFER is not a subtype of $BUFFER. So, a static check can guarantee safe substitution with respect to synchronization.

The calling sequence of objects of type $BUFFER is the regular expression (put get)*, for $FUTURE_BUFFER it is put (get)*. The analogy to finite state machines is obvious. This provides a powerful way of reasoning about the behavior. It also gives means to check subtyping efficiently.

However, a programmer might require a synchronization behavior that finite state machines cannot capture. Furthermore, he might wish to avoid coding each state and prefer to collapse related states into one. Both lead to synchronization behaviors not being entirely captured by the interface. To express such behaviors we extend properties.

5.3 Extended Properties

After introducing the single guarded methods and basic properties, the next step is extending properties. This includes the introduction of the new boolean literal '?' to be used in postconditions. The extension weakens the safety but increases expressibility. It does not affect the high performance that can be achieved with the basic version.

Syntax and Semantics

Properties may also be boolean methods. Like basic properties, these methods must have visitor semantics but do not block. They do not have any parameters, preconditions, or postconditions and their return type is BOOL.

property_definition ⇒ *...* | property *property_identifier* : BOOL is *statement_list* end;

Obviously, extended properties cannot be set by signaling postconditions, but they may be set by side effects of mutator methods. So far, the corresponding state transitions are not visible at the interface. To indicate at least, that the property might change, the syntax of signaling postconditions is extended.

initial_property_expression ⇒ *...* | ?

The value of the boolean literal '?' is always unknown and comparisons with it are always true. It may be used in signaling postconditions to express that the value of an extended property might have changed. Each method that possibly changes an extended property must have a corresponding term in its postcondition stating this possible change. This does not imply that each method changing an attribute must execute such a statement for all extended properties depending on this attribute. So, the code may specify the real postcondition of a method incompletely concerning the synchronization behavior.

Since property comparisons with ? are always true, they have not effect on substitutability and are not considered by subtyping rules. A property comparison with ? in a superclass is implied by any postcondition in the subclass and a property comparison with ? in a subclass does not imply anything in the superclass. Hence, property comparisons with ? are not a specification of the synchronization behavior of the corresponding method, but an operation belonging to the implementation of an interface, that re-evaluates a property, if guards of blocked threads depend on it.

Discussion

By extending properties we generalize the synchronization behavior from a deterministic finite state machine to a non-deterministic one. In the absence of extended properties, signaling postconditions specify a unique transition of the synchronization state, but a comparison with ? in a postcondition describes two possible transitions of a state variable and hence, doubles the number of possible transitions of the synchronization state. The interface is no longer a complete description of the synchronization behavior, because the signaling postcondition may be incomplete. changes of extended properties may be specified by the implementation of a method.

So, if extended properties are used, substitutability of synchronization behavior is not guaranteed by the language, but must be ensured by the implementation, as it is the case for semantic substitutability of subclasses. Extended methods weaken safety, but increase expressibility as the example of a bounded LIFO buffer demonstrates.

```
abstract shield class $LIFO{T} is
   property empty,full:BOOL;
   create(capacity:INT):SAME;
   mutator put(item:T) blocking_pre ~full signaling_post full = ? and empty = ?;
   mutator get:T blocking_pre ~empty signaling_post full = ? and empty = ?;
end;
```

```
shield class LIFO{T} < $LIFO{T} is
    attr buffer:ARRAY{T};
    attr capacity,counter:INT;
    property full:BOOL is return counter = capacity;
    property empty:BOOL is return counter = 0 end;

    create(capacity:INT):SAME is
        r ::= new;
        r.buffer := #(capacity);
        r.capacity := capacity;
        r.counter := 0;
        return r;
    end;

    mutator put(item:T) blocking_pre ~full signaling_post full = ? and empty = ? is
        buffer[counter] := item;
        counter := counter + 1;
    end;

    mutator get:T blocking_pre ~empty signaling_post full = ? and empty = ? is
        counter := counter - 1;
        return buffer[counter];
    end;
end;
```

5.4 Disjunctive Guarded Methods

Our second extension of single guarded methods and basic properties also increases the expressibility of the language, but without weakening safety as the first extension did. Also performance is not affected. Disjunctive guarded methods provide a way to handle threads that are blocked or going to be blocked by a guarded method.

Syntax and Semantics

Guarded methods of an object with signatures differing in their blocking precondition and signaling postcondition only are called *disjunctive guarded methods*. The blocking preconditions determine dynamically which method is chosen, if a call complying with the common part of the signature occurs. The blocking preconditions must be disjoint.

For subtyping, disjunctive methods are one method. The blocking precondition of this method is the disjunction of all disjunctive methods. Its signaling postcondition is a conjunction of precondition - postcondition implications.

Formally: Let m be a method consisting of n disjunctive methods m_i each with the blocking preconditions pre_i and the signaling postconditions $post_i$. The blocking precondition pre_m of m is defined by:

$$pre_m = pre_1 \vee pre_2 \vee \ldots$$

The blocking postcondition $post_m$. of m is defined by:

$$post_m = (initial(pre_1) \Rightarrow post_1) \wedge (initial(pre_2) \Rightarrow post_2) \wedge \ \ldots \ .$$

Discussion

Disjunctive guarded methods help us to cope with problems arising when a thread is blocked or is to be blocked. These problems does not occur in sequential programs, they are caused by the introduction of guarded methods.

For example, we might want to initiate an action in synchronization with other threads, but react with an alternative, if this is not possible. The means guarded methods give us, allow only to block unconditionally until synchronization is established. This is demonstrated by class $MUTEX_NO_TRY implementing a synchronization primitive that is commonly used to provide mutual exclusion. But differently to common designs as POSIX threads [17] and Solaris Threads [41], there is no way to implement a method trylock which acts like method lock but returns immediately, if the blocking precondition is not fulfilled.

```
shield class MUTEX_NO_TRY is
    property locked:BOOL;
    create:SAME is return new.init; end;
    init:SAME signaling_post locked; blocking_pre is return self; end;
    mutator lock is blocking_pre ~locked signaling_post locked=true is end;
    mutator unlock is blocking_pre locked signaling_post locked=false is end;
end;
```

With disjunctive guarded methods we can add a method trylock that conditionally reacts on the value of property locked. The following example shows an implementation.

```
shield class MUTEX is
    property locked:BOOL;
    create:SAME is return new.init; end;
    init:SAME blocking_post locked=false is return self; end;
    mutator lock blocking_pre ~locked signaling_post locked=true is end;
    mutator trylock:BOOL
        blocking_pre ~locked signaling_post locked=true is
        return true end;
    trylock:BOOL blocking_pre locked is return false; end;
    mutator unlock blocking_pre locked signaling_post locked=false is end;
end;
```

In general, disjunctive guarded methods offer a way to switch on synchronization events. This allows not only trying a guard as in the example above, but also more complex applications, e.g. multiplexing of asynchronously incoming messages and save termination of blocked threads. Class MUTEX_WITH_TERMINATION demonstrates safe termination of blocked threads.

```
shield class MUTEX_WITH_TERMINATION is
    property locked:BOOL;
    property terminated:BOOL;
    create:SAME is return new.init; end;
    init:SAME blocking_post locked=false and terminated=false is return self; end;
    mutator lock blocking_pre ~locked signaling_post locked=true is end;
    mutator lock blocking_pre terminated is raise termination_exception end;
    mutator trylock:BOOL
        blocking_pre ~locked signaling_post locked=true is
        return true end;
    trylock:BOOL blocking_pre locked is return false; end;
    mutator unlock blocking_pre locked signaling_post locked=false is end;
end;
```

It extends class MUTEX by two lines, the declaration of property terminated and an alternative method for lock that unblocks all threads waiting to lock and raises an exception to handle termination.

5.5 Implementation

Guarded Methods have been designed for high performance applications. This section demonstrates how they can be implemented efficiently. We suggest a common synchronization scheme for guarded methods and reader/writer protection that requires no context switches for guard re-evaluation and that helps reducing the number of necessary re-evaluations.

5.5.1 Integration with reader/writer synchronization

The synchronization required to implement guarded methods can be integrated with the reader/writer synchronization already required by the object model. Since the synchronization points are exactly the same, a common scheme can be used to implement both.

We suggest a simple scheme that efficiently realizes reader/writer protection and method guards:

1. Before a thread enters a somehow protected method, it tries to satisfy all synchronization constraints and either enters the method or blocks itself. The synchronization constraints may contain reader/writer synchronization for the object the method is called on, reader/writer synchronization for arguments, and the guard of the called method.
2. After a thread leaves a protected method, it checks all blocked threads that might satisfy their synchronization constraints because the current thread has left the method, and unblocks all threads that do so.

Integration of both synchronization mechanisms is not really necessary but intended by the design of guarded methods. It simplifies the run-time system and has no negative impact on performance.

5.5.2 Avoiding Context Switches

Since properties are attributes or methods of the shield object, the re-evaluation of a guard can be executed by threads other than the blocked one. Hence, a thread leaving a method and executing step 2 of the scheme above can exactly determine which of the blocked threads satisfies its synchronization constraints and unblock those without any context switch. We mention this, because it distinguishes Sather 2 from languages like Java which require each thread to re-evaluate a guard or a similar condition variable itself. This procedure produces a significant overhead of context switching and synchronization.

5.5.3 Reducing the Number of Guard Re-Evaluations

Dependencies between blocking preconditions and signaling postconditions can be checked statically. For each pair of these one of three kinds of dependencies can be determined:

1. A signaling postcondition implies a blocking precondition to be true.
2. A signaling postcondition implies a blocking precondition to be false.
3. A signaling postcondition might affect a blocking precondition.
4. A signaling postcondition does not affect a blocking precondition.

For dependencies 1, 2, and 4, a thread leaving a guarded method does not have to re-evaluate the guard of the corresponding blocked thread in order to decide, whether this thread can be unblocked. Only for dependency 3, the guard has to be re-evaluated.

Our design allows to reduce the number of pairs with dependency 3 by static analysis. Since blocking preconditions are restricted to expressions on properties and signaling postconditions specify exactly which properties might be affected and which not, static analysis can reduce the number of these pairs close to the minimum.

This analysis, checking blocking preconditions and signaling postconditions only, is quiet simple. In general, even a further reduction is possible, in particular when '?' is used in a postcondition. But this analysis would include the method bodies and might be significantly more complex.

5.5.4 Deadlock Detection

For deadlock detection we suggest static and dynamic checks. A simple deadlock situation that can be checked statically occurs if a visitor or a mutator method calls a guarded method on the same object. If such a call is blocked because of a property of this object, it can never be unblocked, since it still is a visitor or mutator and no other thread can change any property of the object.

Further static deadlock detection can be based on either the dependency graph of guarded methods or the property transition graph. The dependency graph contains a node for each guarded method and an edge for each dependency of type 1, 2, or 3 defined in the previous section. The property transition graph of an object with n properties contains 2^n nodes representing property states and edges representing transitions of the property state. A guarded method defines transitions for all property states satisfying the blocking precondition. In absence of a '?' in the signaling postcondition, one transition per property is defined by a method. Each '?' doubles the number of transitions.

Each of these graphs describes all legal calling sequences on a shield object. Deadlock detection can be based on the comparison of the legal sequences with the calling sequences of a program. But since in general, a calling sequences of a program can be determined only partially, this detection of deadlocks is restricted.

Different to static checks, dynamic deadlock detection does not prevent deadlocks, but is a debugging tool that may provide helpful run-time information. While static deadlock detection is mainly used for intra-object deadlocks, dynamic deadlock detection is most important for inter-object deadlocks, e.g. two threads each already being visitor of an object

and each being blocked by a guard of the other object, respectively. This deadlock can never be solved, because only a writer may change properties and no writers can access the objects as long as there are (blocked) readers. This problem of nested synchronization is not introduced by guarded methods, it already occurs with the object model, e.g. two threads each already being mutator of an object and each trying to become a visitor of the other object, respectively.

6 Data-Parallel Features

The following section describes the data-parallel features of the language design. For convenience, the data-parallel features are described in terms of syntactic sugar over the basic object model; however, they make use of unprotected attributes, since protection is ensured by other means.

6.1 Definition of pardo and syncdo statements

Threads can be created explicitly with the *pardo* or the *syncdo* statement. They consist of a header and a body.

statements ⇒ ... | *pardo* | *syncdo*
pardo ⇒ for all *header* do in parallel *body* end
syncdo ⇒ for all *header* do in synchrony *body* end

The header contains an identifier v and either an expression of an array type a or iterator. The array and the iterator, resp., must have an element type and a return type, resp., assignable the type of v. The pardo and the syncdo statement create a.asize threads and as many threads as the iterator object can be called without breaking, respectively.

header ⇒ *identifier* in *iterator* | *expression*

All threads share the concurrent attributes of the object and concurrent parameters and concurrent local variables of the method the pardo or the syncdo statement occurs in. In thread *i*, v is assigned the value of a[i] or the return value of the *i*-th call to iter. Variables defined in the body of the pardo or syncdo statements are local to each thread. All threads execute the code specified in the body.

body ⇒ *statement_list*

All threads of a pardo statement run asynchronously. All threads of a syncdo statement run in lock step manner, i.e, a barrier synchronization occurs after each read from and each write to concurrent variables.

A thread terminates if it has executed its last statement. A pardo or the syncdo statement is finished if all its threads terminated.

Remarks

Creating threads with an iterator in general requires linear time since the iterator has to be called sequentially. The creation of threads with an index array can be implemented in logarithmic time. In practice, the optimal broadcast tree technique [21] can be applied to improve the speed compared to the iterator version.

Many parallel algorithms are designed in PRAM [20] like style. In this special case, the variable v is of type INT and the values of v for the single threads ranges from some lower upto some upper bound. It may be implemented by an iterator lower_bound.upto!(upper_bound) in the pardo or syncdo header. This notation does not differ in the time complexity for creating threads since upto is a build-in iterator in the build-in class INT. It cannot be changed. Hence the number of threads and their values v are determined if the lower and the upper bound are determined. This permits the same fast thread creation which not possible for general iterators.

In general a threads identification v is not an integer and the shared data structure is not an array, see example programs below. Furthermore, we don't restrict to synchronous execution of parallel threads.

6.2 Pardo and Syncdo Statements as Syntactic Sugar

The features described above may be implemented in terms of the basic parallel constructs. This section describes a naive implementation that works correctly. For optimizations we refer to techniques described e.g. in [49][27][28] that remove synchronization barriers and distribute the shared data structures.

For each pardo or syncdo statement s, we create a method m_s and a fork statement f_s. The signature of m_s declares parameters conforming to concurrent attributes of the class and the concurrent parameters and concurrent local variables of the method containing s. If s is a pardo statement, the body of m_s equals the body of the s except for an additional barrier synchronization at the end of the m_s's body. If s is a syncdo statement, the body of m_s equals the body of s except for additional barrier synchronizations after each read or write access to m_s's parameters and at the end of m_s's body.

The number of forks f_s on m_s guarantees the correct number of threads. Each f_s passes the concurrent attributes, concurrent parameters, and concurrent local variables as parameters to m_s. Additionally, it passes an object that handles the barrier synchronization. s is replaced by the f_ss .

There is a build-in class that handles the barrier synchronization called BARRIER. Objects of this class are created and initializes at the beginning of m_s. Whenever a barrier synchronization is required, the method synchronize is called by the threads.

Examples

The following method copies the upper right triangle matrix to the lower left triangle matrix.

```
matrix ::= # SHIELD_ARRAY2 {INT}(10,10);

copy_triangle_matrix is
    i,j : INT;
    for i in 0.upto!(9) do in parallel
        for j in 0.upto!(9) do in parallel
```

```
            matrix[i,j] := matrix[j,i];
        end;
    end;
end; -- copy_triangle_matrix
```

This program is the translated according to the naive implementation.

```
matrix ::= # SHIELD_ARRAY2 {INT}(10,10);

copy_triangle_matrix' is
    b::=# BARRIER;
    loop
        i := 0.upto!(9)
            fork(parloop1(matrix,i,b));
    end;
    b.init(11);
    b.synchronize;
end; -- copy_triangle_matrix'

parloop1(matrix : SHIELD_ARRAY2{INT}; i : INT; barrier: BARRIER is
    b ::= #BARRIER
    j : INT;
    loop
        j := i.upto!(9);
        fork(parloop2(matrix,i,j,b));
    end;
    b.init(11-i);
    b.synchronize;
    barrier.synchronize;
end; -- parloop1

parloop2(matrix : SHIELD_ARRAY2{INT}; i,j : INT; barrier: BARRIER is
    matrix[i,j] := matrix[j,i];
    barrier.synchronize;
end; -- parloop2
```

The next examples implement the algorithm of pointer jumping on arrays and lists. The threads in the first two examples execute their programs in lock-step manner while in third example all threads work asynchronously.

```
a ::= # SHIELD_ARRAY{INT};
pointer_jumping is
    i,j : INT;
    for i in 0.upto!(n-1) do in synchrony
        loop j :=1.upto!(n.log.ceil);
            a[i] := a[a[i]];
        end;
    end;
end; -- pointer_jumping


anchor : SHIELD_LINKED_LIST{T};
pointer_jumping' is
    j : INT;
    list_node : SHIELD_ LINKED_LIST{T};
    for list_node in anchor.elts! do in synchrony
        loop j :=1.upto!(anchor.size.log.ceil);
            list_node.next := list_node.next.next;
        end;
    end;
end; -- pointer_jumping'
```

```
pointer_jumping'' is
    j : INT;
    list_node : SHIELD_ LINKED_LIST{T};
    for list_node in anchor.elts! do in parallel
        while list_node.next /= list_node.next.next loop
            list_node.next := list_node.next.next;
        end;
    end;
end; -- pointer_jumping''
```

Appendix A: Shield Bag (Abstract Class)

```
abstract shield class $SHIELD_BAG{ETP} < $RO_BAG{ETP}, $VAR
-- An unordered container in which the elements are not unique.
--
-- This is a reference abstraction and supports operations that modify
-- self. Instances of subtypes may be viewed as variables with a value
-- of $VBAG{ETP}
--
-- For pointers to other documentation please see the class comment in
-- the read-only abstraction $RO_BAG
--

is

    mutable visitable as_value:$VBAG{ETP};
    -- Return the current value associated with self

    mutable add(visitable able e:ETP);
    -- Add the element ‘e’ to self
    -- self <- initial(self).add(e)

    mutable delete(visiitable e:ETP);
    -- Delete at most one occurance of ‘e’ from self
    -- self <- initial(self).delete(e)

    mutable delete_all(visitable e:ETP);
    -- Delete all occurrences of ‘e’ from self
    -- self <- initial(self).delete(e)

    mutable clear;
    -- Delete all elements of self. post result.size = 0

    mutable to_concat(mutable arg:$RO_BAG{ETP});
    -- Concatenate the elemetns of ‘arg’ to this bag
    -- self <- initial(self).add_bag(arg)

    mutable to_union(visitable arg: $RO_BAG{ETP});
    -- Turn this bag into the union of self and ‘arg’
    -- self <- initial(self).union(arg)

    mutable to_intersection(visitable arg:$RO_BAG{ETP});
    -- Turn this bag into the intersection of self and ‘arg’
    -- self <- initial(self).intersection(arg)

    mutable add(visitable e:ETP):$SHIELD_BAG{ETP};
    -- Result is a new bag containing all the elements of self and ‘e’

    mutable delete(visitable e:ETP):$SHIELD_BAG{ETP};
    -- Result is a new bag containing all the elements of self except for
    -- an element equal to ‘e’, if one exists. If more than one element
    -- is equal to ‘e’, delete only one of them

    mutable delete_all(visitable e:ETP):$SHIELD_BAG{ETP};
    -- Result is a new bag containing all the elements of self except for
    -- any elements equal to ‘e’
```

```
visitable count(visitable e:ETP):INT;
-- Return the number of occurences of ‘e’ in self

visitable unique!:ETP;
-- Yield the unique elements of self. Equivalent to self.as_set.elt!

mutable n_unique: INT;
-- Returns the number of unique elements in the bag
--
-- result = number of unique elements

visitable is_subset_of(visitable arg: $RO_BAG{ETP}): BOOL;
-- Returns true if ‘self’ is a subset of ‘arg’. For elements that occur
-- multiple times, the number of occurences of the element in ‘arg’
-- must be greater than or equal to the number of occurences in self
--
-- result=true iff for all e in self: count(e) <= arg.count(e)

visitable concat(visitable arg:$ELT{ETP}): $RO_BAG{ETP};
-- Returns a bag containing all the elements of self and ‘arg’.
-- For elements that occur multiple times, the result contains
-- the sum of the number of occurences in self and ‘arg’
--
-- result=bag of all e s.t. result.count(e)=self.count(e)+arg.count(e) > 0

visitable union(visitable arg: $RO_BAG{ETP}): $RO_BAG{ETP};
-- Returns a bag containing the elements of ‘self’ and ‘arg’.
-- For elements that occur multiple times, the result contains
-- the maximum number of occurences in either self or ‘arg’
-- This definition permits the union of sets to be consistent
-- with the union of bags.
--
-- result=bag of all e s.t.
--         result.count(e)=max(self.count(e),arg.count(e)) > 0

visitable intersection(visitable arg: $RO_BAG{ETP}):$RO_BAG{ETP};
-- Returns a bag containing the elements common to self and ‘arg’
-- For elements that occur multiple times, the result contains
-- the minimum number of occurrences in either self or ‘arg’
--
-- result=bag of all e s.t.
--          result.count(e)=min(self.count(e),arg.count(e)) > 0

visitable is_empty:BOOL;
-- Returns true if the size of the container = 0

mutable size: INT;
-- Number of elements contained

visitable copy: SAME;
-- Return a copy of the current container

visitable has(visitable e: ETP): BOOL;
-- True if the container contains the element “e”

visitable equals(visitable c:$RO_BAG{ETP}):BOOL;
-- Return true if both containers contain the same elements with
-- the same number of repetitions, irrespective of the order of the
-- elements

visitable as_array:ARRAY{ETP};
-- Return the elements of the container in an array
```

```
    visitable elt!:ETP;
    -- Yield all the elements of self. The order is not defined.

    visitable str:STR;
    -- Yield a string version of self

end; -- $SHIELD_BAG{ETP}
```

References

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

[2] 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.

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

[4] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

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

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

[7] G.R. Andrews et al., An overview of SR language and implementation, ACM transactions of Programming Languages and Systems 10(1):51-86, January 1988.

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

[9] 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.

[10] D. Decouchant et al., A synchronization mechanism for typed objects in a distributed system, in: Proceedings of the 1988 ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices 24:105-107, ACM Press, April 1989.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] D.G. Kafura, K.H. Lee, Inheritance in Actor based concurrent object-oriented languages, in: Proceedings of ECOOP'89, Cambridge University Press, 1989, pp. 131-145.

[20] R.M. Karp, V. Ramachandran, Parallel algorithms for shared memory machines. In Handbook of theoretical Computer Science Vol. A, pp. 871-941. MIT-Press, 1990.

[21] R.M. Karp, A. Sahay, E.E. Santos, K.E. Schauser, Optimal Broadcast and Summation in the logp Model. ACM Symposium on Parallel Algorithms and Architectures, 1993.

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

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

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

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

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

[27] W. Löwe, W. Zimmermann, On finding optimal clusterings of task graphs. In Proceedings of the First Aizu International Symposium on Parallel Algorithms and Architecture Synthesis, pp. 241-247. IEEE Computer Society Press, 1995.

[28] W. Löwe, W. Zimmermann, J. Eisenbiegler, Optimization of Parallel Programs on Machines with Expensive Communication. Will appear in Proceedings EUROPAR'96, Springer-Verlag, 1996.

[29] S.E. Lucco, Parallel Programming in a virtual object space, in: Proceedings of OOPSLA'87, SIGPLAN Notices 22:26-34, ACM Press, October 1987.

[30] S. Matsuoka, A. Yonezawa, Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, in Research Directions in Concurrent Object-Oriented Programming, eds. G. Agha, P. Wegner, and A. Yonezawa. The MIT Press, 1993. pp. 107-150.

[31] 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.

[32] B. Meyer, Object-oriented software construction, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[33] 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.

[34] 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.

[35] 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.

[36] 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

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

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

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

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

[41] Sun Microsystems, Solaris Multithreaded Programming Guide, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

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

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

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

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

[46] C. Tomlinson, V. Singh, Inheritance and synchronization with Enabled-sets, in: Proceedings of OOPSLA'89, SIGPLAN Notices 24:103-112, ACM Press, October 1989.

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

[48] 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)

[49] W. Zimmermann, W. Löwe, An Approach to Machine-Independent Parallel Programming. LNCS 854, Parallel Processing: CONPAR'94 - VAPP VI, pp. 277-288, Springer-Verlag, 1994.