Minimal Ownership for Active Objects *

Dave Clarke[1], Tobias Wrigstad[2], Johan Östlund[2], and Einar Broch Johnsen[3]

[1]CWI, Amsterdam, The Netherlands
[2]Purdue University, USA
[3]University of Oslo, Norway

Abstract Active objects offer a structured approach to concurrency, encapsulating both *unshared* state and a thread of control. For efficient data transfer, data should be passed by reference whenever possible, but this introduces aliasing and undermines the validity of the active objects. This paper proposes a minimal variant of ownership types that preserves the required *race freedom* invariant yet enables data transfer by reference between active objects (that is, without copying) in many cases, and a cheap clone operation where copying is necessary. Our approach is general and should be adaptable to several existing active object systems.

1 Introduction

Active objects have been proposed as an approach to concurrency that blends naturally with object-oriented programming [1,37,61]. Several slightly differently flavoured active object systems exist for Java [8], Eiffel [17, 46], C++ [43] et al. Active objects encapsulate not only their state and methods, but also a single (active) thread of control. Additional mechanisms, such as *asynchronous method calls* and *futures*, reduce the temporal coupling between the caller and callee of a method. Together, these mechanisms offer a large degree of potential concurrency for deployment on multi-core or distributed architectures.

Internal data structures of active objects, used to store or transfer local data, do not need independent threads of control. In contrast to the active objects, these *passive objects* resemble ordinary (Java) objects. An immediate benefit of distinguishing active and passive objects is that all the concurrency control is handled by the active objects, and locking (via synchronised methods) becomes redundant in the passive objects. This simplifies programming and enables the (re-)use of standard APIs without additional concurrency considerations.

Unfortunately, introducing passive objects into the model gives rise to aliasing problems whenever a passive object can be shared between more than one active object. This allows concurrent modification and/or observation of changes to the passive data objects. Specifically, two active objects can access the same passive data; if at least one thread modifies the data, then different access orders may produce different results unless we re-introduce locks into the programming

* This work is in the context of the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (`http://credo.cwi.nl`).

model. The resulting system would be as difficult to reason about as unconstrained shared variable concurrency. This problem can be addressed several ways, neither of which we feel is entirely satisfactory:

Immutable Data Only Active objects are mutable, but field values belong to immutable data types; e.g., integers or booleans, immutable objects such as Java-style strings or XML, or Erlang- and Haskell-style datatypes [5, 35].

Cloning Locally, active objects can arbitrarily access passive objects, but when data is passed between active objects, the data must be deeply cloned. This approach is taken for distributed active objects (e.g., [8, 18]).

Unique References Only one reference to any passive object is allowed at any time. Passive objects can be safely transferred between active objects.

Emerald [33, 51] partly addresses this problem using the first approach. Objects can be declared immutable to simplify sharing and for compiler optimisation, but immutability is an unchecked annotation which may be violated. Emerald's use of immutability is optional, as adopting pure immutability means that programs can no longer be implemented in an imperative object-oriented style.

ProActive [8] uses the second approach and copies all message parameters. The programmer gets a very simple and straightforward programming model, but the copying overhead is huge in message-intensive applications.

Last, using uniqueness requires a radical change in programming style and may result in fragile code in situations not easily modelled without aliasing.

This paper investigates the application of ownership types in the context of active object-based concurrency. We propose a combination of ownership-based techniques that can identify the boundaries of active objects and statically verify where reference semantics can be used in place of copy semantics for method arguments and returns.

In previous work, we combined ownership types with effects to facilitate reasoning about disjointness [20] and with uniqueness to enable ownership transfer [21]. Recently, we coalesced these to realise flexible forms of immutability and read-only references [49]. In this paper we tune these systems to the active objects concurrent setting and extend the resulting system with the *arg* reference mode from Flexible Alias Protection [48]. Furthermore, our specific choices of ownership defaults make the proposed language design very concise in terms of additional type annotations. The main contributions of this paper are:

- A synthesised minimal type system with little syntactic overhead that identifies active object boundaries. This type system enables expressing and *statically checking* (and subsequent compiler optimisations) safe practices that programmers do manually today (framework permitting), such as:
  - Statically guarantee total isolation of active objects;
  - In a local setting, replace deep copying of messages with reference passing for (parts of) immutable objects, or unique objects;
  - In a distributed setting replace remote references to immutable (parts of) objects with copying for more efficient local access; and

    * Immutability is per object and the same class can be used to instantiate both immutable and mutable objects.

  All necessary annotations are expressed in terms of ownership, plus a trivial effects system which makes the formalisation (see [22]) clean and simple.
- We present our system in the context of a Java-like language, Joëlle, however our results are applicable to any active object or actor based concurrency model. Active object systems such as ProActive [8], Emerald [33, 51], and Scoop [41] use unchecked immutability or active annotations. Integrating our type system with these approaches for static checking seems straightforward.

The formal description of the system, which is a synthesized model of a large body of previous work [20, 21, 25, 49, 59], can be found in [22].

Organisation Section 2 surveys the alias control mechanisms upon which we build our proposal. Section 3 further details the problem and presents our solution. Section 4 compares our work with related work, and Section 5 concludes.

2 Building Blocks

We now survey the alias control mechanisms used to construct our synthesized system. They address the problem of reasoning about *shared mutable state* [31, 48], which is problematic as a shared object's state can change *unexpectedly*, potentially violating a sharer's invariants or a client's expectations. There are three main approaches to this problem:

ownership: encapsulate all references to an object within some *box*; such as another object, a stack frame, a thread, a package, a class, or an active object [3, 4, 11, 13, 19, 23, 30, 45, 48].

uniqueness: eliminate sharing so that there is only one active reference to an object [3, 11, 14, 21, 30, 42].

immutability: eliminate or restrict mutability so an object cannot change, or so that changes to it cannot be observed [10, 15, 48, 55, 58, 62].

2.1 Ownership

Ownership types [23] initially formalised the core of Flexible Alias Protection [48]; variants have later been devised for a range of applications [3,11,13,19,23,45,48]. In general, object graphs form an unstructured "soup" of objects. Ownership types impose structure on these graphs by first *putting objects into boxes* [27], then imposing a topology [2,19] on the boxes, and finally restricting the way objects in different boxes can access each other, either prohibiting certain references or limiting how the references can be used [44, 45].

Ownership types record the box in which an object resides, called the *owner*, in the object's type. The type system syntactically ensures that fields and methods with types containing the name of a private box are encapsulated (thus

only accessible by `this`). This encapsulation ensures that the contents of private boxes cannot be exported outside their owner. For this to work, the owner information must be retained in the type. Consider the following code fragment:[1]

```
class Engine {} class Car { this::Engine e; }
```

In class `Car`, the owner of the `Engine` object is `this`, which indicates that the object in the field `e` is owned by the current instance of `Car` (or, in other words, that every car has its own engine). The type system ensures that the field `e` is accessible only by `this`, the owning object.

Ownership types enforce a constraint on the structure of object graphs called *owners-as-dominators*. This property ensures that access to an object's internal state goes through the object's interface: the only way for a client of a `Car` object to manipulate the `Car`'s `Engine` is via some method exposed in the `Car`'s public interface. Some ownership types proposals [2, 3, 12, 45] weaken this property.

All classes, such as `Engine` above, have an implicit parameter `owner` which refers to the owner of each instance of the class. Thus, arbitrary and extensible linked data structures may be encapsulated in an object. Contrast this with Eiffel's expanded types [40] and C++'s value objects [57], which enable an object to be encapsulated in another object, but require a fixed sized object. In the following class

```
class Link { owner::Link next; int data; }
```

the `next` object has the same owner as the present object. This is a common idiom, and we call such objects *siblings*. (The Universes system [45] uses the keyword `peer` instead of `owner`.)

2.2 External Uniqueness

Object sharing can be avoided using unique or linear references [3, 11, 14, 21, 30, 42]: at any point in the execution of a program, only one *accessible* reference to an object exists. Clarke and Wrigstad introduced the notion of *external uniqueness* [21,59] which fits nicely with ownership types and permits unique references to aggregate objects that are inherently aliased, such as circularly linked lists. In external uniqueness, unique references must be (temporarily) made non-unique to access or call methods on fields. The single external reference is thus the only active reference making the aggregate effectively unique. External uniqueness enables ownership transfer in ownership types systems.

External uniqueness is effectively equivalent to introducing an owner for the field or variable holding a reference into the data structure, such that the only occurrence of that owner is in the type of the field or variable. In the code below, `first` holds the only pointer to the (sibling) link objects.

```
class List { unique::Link first; }
```

Uniqueness can be maintained with e.g., destructive reads or Alias Burying [14].

[1] In this section, code uses syntax from Joe-like languages [20, 21, 49].

In summary, ownership deals with encapsulating an entire aggregate. Uniqueness concerns having a single (usable) reference to an object. External uniqueness combines them, resulting in a single (usable) reference to an entire aggregate.

2.3 Immutability and 'Safe' Methods

Immutable objects can never change after they are created. An *immutable reference* prevents the holder from calling methods that mutate the target object. Furthermore, references to representation objects returned from a method call via an immutable reference are also immutable—or immutability would be lost. *Observational exposure* [15] occurs when an immutable reference can be used to observe changes to an object, which is possible if non-immutable aliases exist to the object or its representation. Fortunately, strong encapsulation, uniqueness, and read-only methods make the (staged[2]) creation of "truly immutable" objects straightforward [49]. This is similar to Fähndrich and Xia's Delayed Types [28].

In Flexible Alias Protection [48], '*arg*' or *safe references* (our preferred terminology) to an object may only access immutable parts of the object; i.e., the parts which do not change after initialisation. Thus, clients accessing an object via a safe reference can only depend on the object's immutable state, which is safe as it cannot change unexpectedly. Safe references can refer to *any* object, even one which is being mutated by a different active object, without any risk of observational exposure.

2.4 Owner-Polymorphic Methods

Owner-polymorphism is crucial for code reuse and flexibility in the ownership types setting [19,59]. *Owner-polymorphic methods* are parameterised with owners to give the receiver temporary permission to reference an argument object. For example, the following method accepts an owner parameter `foo` in order to enable a list owned by any other object to be passed as an argument:

```
<foo> int sum(foo::List values) { ... }
```

Clarke [19] established that owner-polymorphic methods can express a notion of *borrowing*: an object may be passed to another object, which does not own it, without the latter being able to capture a reference to the former. (For further details, see [59].) Owner-polymorphic methods are reminiscent of region-polymorphic procedures in Cyclone [29].

3 Active Ownership

In *Concurrent programming in Java* [38], Doug Lea writes that "To guarantee safety in a concurrent system, you must ensure that *all* objects accessible from multiple threads are either immutable or employ appropriate synchronisation,

[2] In *staged object creation* an object is initialised though a series of method calls, potentially within different objects, rather than exclusively in its constructor [49].

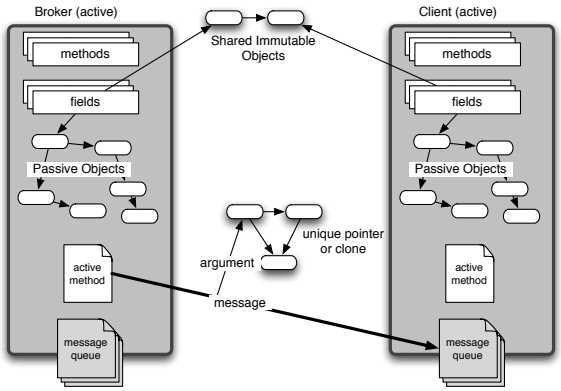

Figure 1. Active Ownership. Safe references are not depicted. Broker and Client are active objects from the code example in Figure 2.

and also must ensure that *no* other object ever becomes concurrently accessible by leaking out of its ownership domain."

The simple ways to guarantee the above are the first two we listed on Page 2: making everything immutable or use deep copying semantics. While efficient, the first is very restrictive and requires careful inspection of the code to determine that the messages are truly immutable. The second is easier to check (just clone all arguments) but has the downside of adding massive copying overhead.

We argue that the most effective approach is a pragmatic combination: using unique references or immutable objects where possible and deep copying only as a last resort. To enable this in a statically and modularly checkable fashion, we introduce a few extra annotations on interfaces of active object classes. We believe that requiring these extra annotations will be helpful for maintenance and possibly also refactoring. Most importantly, we believe that the static checking enabled by the annotations will save time, both programmer-time and run-time.

Our alias control mechanisms uphold the invariant that *no two 'threads' concurrently change or observe changes to an object,* which is the invariant obtained by the deep copying of message arguments in ProActive (with minor exceptions unimportant to us here). Note that our type system is agnostic of the thread model—it correctly infers the boundaries between active objects regardless.

3.1 Active and Passive Classes

Active and passive objects are instantiated from active and passive classes. Active classes are declared with the `active`. Passive is default.

Active objects encapsulate a *single* thread of control. They primarily interact via asynchronous method calls that return future values. A future is a placeholder for a value which need not be currently available [9]. For asynchronous calls, the future is the placeholder for the methods' actual return values. Thus, the caller need not wait for the call to complete. A future's value is accessed using its `get`

```
active class Client {
   void run() {
      Request rm = ...; // formulate Request
      future Offer offer = myBroker!book(rm.clone()); // †
      ... // evaluate offer
      offer.getProvider()!accept(offer.clone()); // †
   }
}
active class Broker {
   void run() { ... } // go into reactive mode
   // book returns first Offer that responds to the request
   Offer book(Request request) { ... }
}
active class Provider {
   void run() { ... } // go into reactive mode
   Offer query(Request request) { ... }
   boolean accept(Offer offer) { ... }
}
class Request {
   Request(String desc) { ... }
   void markAccepted() { ... }
}
class Offer {
   Offer(Details d, Provider p, Request trackback) { ... }
   Provider getProvider() { ... }
}
```

Figure 2. Example of active objects exchanging arguments by copying (shown at †). Here `future Offer` denotes a future of type `Offer`. For clarity, we use a `!`-notation on asynchronous method calls, e.g., `myBroker!book(rm)`.

method, which blocks until the future has a value, which is standard practise. Synchronous calls may be encoded by calling `get` directly after asynchronous method calls. Method calls to passive objects are always synchronous; i.e., they are similar to standard method calls as found in Java. Multiple asynchronous calls to an active object are put in an inbox and executed sequentially. Creol [25, 34] uses release point to yield execution midway through a method, but this is irrelevant to the minimal ownership system.

Figures 1 and 2 show a use of active objects that deliberately copy arguments to avoid aliasing-induced data races. Figure 4 shows how we can annotate the code in our system to avoid the copying without comprising safety. For brevity, we focus simply on the interfaces, which suffices for the type annotations. The figure shows the following scenario:

1. Client sends request to broker
2. Broker forwards request to provider(s) and negotiates a deal
3. Broker returns resulting offer to client
4. If client accepts offer, client sends acceptance to provider

The client, the broker, and all providers are represented as active objects and execute concurrently. In contrast, requests and offers are passive objects, passed between active ones by copying to avoid data races between the active objects.

3.2 Putting it Together: Language Constructs for Active Ownership

This section describes our synthesised system leading up to an encoding of the example from Figure 2 that avoids copying. Figure 3 shows a few examples of code using our annotations. While our annotations are similar to capabilities, they are expressed as owners. This keeps the underlying formalism (see [22]) relatively simple. Our system has the following annotations:

`active`	globally accessible owner of all active objects
`owner`	the owner of the current object (in the scope of the annotation)
`this`	the owner denoting the current object (in the scope of the annotation)
`unique`	the owner denoting the current field or variable
`immutable`	globally accessible owner of all immutable objects
`safe`	globally accessible owner allowing safe access

The owners `active`, `immutable`, and `safe` are available in any context, and denote the global owner of active objects, immutable references, and safe references, respectively. `unique` is also available everywhere, but denotes field-as-owner, as explained in Section 2.2. Nested inside each active object is a collection of passive objects, owned by the active object with owner `this`. The owner `owner` is available only in passive classes for referring to the owner of the current instance, and is used to create data structures within an active object.

Note that the ownership hierarchy is very flat, as there is no owner `this` inside a passive class. Ownership encapsulates passive objects inside an active object. Consequently, there is no need to keep track of nesting or other relationships such as links between owners [2]. In addition, the classes in this system take no owner parameters, in contrast to the original ownership types system [23]. Therefore, no run-time representation of ownership is required [60].

Immutable and Safe References Immutable types have owner `immutable`. In our system, only passive objects can have immutable type. Fields or variables containing immutable references are not final unless explicitly declared final or if the container enclosing the field is immutable. In order to preserve immutability, only *read-only* and *safe* methods (see below) can be called on immutable objects.

Safe references (called *argument* references in Flexible Alias Protection [48]) have owner `safe` and can be used only to access the final fields of an object, and the final fields of the values returned from methods, and so forth. These parts of an object cannot be changed underfoot. Methods that obey these conditions are called safe methods, denoted by a `safe` annotation. Any non-active type can be subsumed into a `safe` type.

Immutable references can only be created from unique references. The operation consumes the unique reference and thus guarantees the absence of any aliases to the object that allows mutation. This is powerful and flexible as it allows a single class to be used both as a template for both mutable and immutable objects (see [21]) and staged object construction. Effectively, immutability becomes a property of the object, rather than of the class or of references.

```
active class Foo { // active class
  this Bar f; // properly encapsulated field
  owner Bar b; // invalid -- owner is not legal in active classes
  active Bar k; // reference to (sibling) active object
}

class Bar { // passive class
  owner Bar f; // sibling field (same level of encapsulation)
  this Bar b; // invalid -- this is not legal in passive classes
}

unique Foo f = new Foo(); // new returns a unique reference
immutable Foo b = f--; // -- is destructive read, nullifies f

void foo() read { ... } // can only call read and safe methods
                        // on this, and not update fields

void foo() safe { ... } // can only call safe methods/read
safe Foo f;             // final immutable/safe fields on this/f

void foo() write { ... } // regular method, write can be omitted
```

Figure 3. Examples of active and passive classes, unique, safe and immutable types, and read and safe methods.

Safe references do not preclude the existence of mutable aliases, which is safe as it only allows access to the referenced object's immutable parts. Consequently, both safe and immutable references avoid observational exposure.

Read-only and Safe Methods Following previous read-only proposals, e.g., [10,15,30,55], a read-only method preserves the immutability of objects, and does not return non-immutable references to otherwise immutable objects. Read-only methods cannot update any object with owner `owner`, which notably includes the receiver. They are not, however, purely functional: they can be used to modify unique references passed in as arguments or objects freshly created within the method itself and they can call mutating methods on active objects.

As immutability is encoded in the owners, a return value from a read-only method that has owner `owner` will (automatically) have the owner `immutable` when the read-only method is called on an immutable reference, and hence will not provide a means for violating the immutability of the original reference [49, 62]. To allow modular checking, read-only methods are annotated with `read`.

A safe method, annotated `safe`, is an additionally restricted read-only method that may only access immutable parts of the receiver's state, i.e., final fields containing safe or immutable references. Conceptually, a read-only method prevents mutation whereas a safe method also prevents the observation of mutation.

3.3 Data Transfer and Minimal Cloning

To ensure that the data race freedom invariant is preserved, care is needed when passing data between active objects. How data is passed, depends on its owner.

`Active`-owned objects are safe to pass by reference as external threads of control never enter them by virtue of asynchronous methods calls and futures. `Immutable` and `safe`-owned objects are obviously safe to pass by reference as their accessible parts cannot be changed. Last, `unique` objects are safe to pass by reference as they are effectively transferred to the target[3].

Other objects (owned by `this`, `owner` and owner-parameters to methods) must be cloned. Cloning returns a unique reference which can be transferred regardless of the owner of the expected parameter type. Cloning follows the above rules to determine whether an object's fields must be cloned or whether it is safe to simply copy the reference—the only difference is that clone clones fields holding `unique` references. A "minimal clone" operation can be trivially inferred statically from the owner annotations. This is similar to the *sheep clone* described for ownership types [19, 47] and Nienaltowski's object import [46].

Notably, our clone rightfully avoids cloning of active objects, something a naive clone would not do. This is necessary to allow returning a reference to a provider in our example and lets active objects behave like regular objects.

Reducing Syntactic Baggage We adopt a number of reasonable defaults for owner annotations to reduce the amount of annotations required in a program, and to use legacy code immediately in a sensible way.

Passive Classes (including all library code) have one implicit owner parameter `owner`, which is the default owner of all fields and all method arguments. Note that this means that library code, in general, requires no annotations.

Active Classes have the implicit owner `active`. In an active class, the default owner is `this` for all fields and `unique` for all method arguments.

Together these defaults imply that all passive objects reachable from an active object's fields are encapsulated inside the active object, in the absence of immutable and safe references. By default, all method parameters in the public interface of active objects are `unique`. This is the only way to guarantee that *mutable* objects are not shared between active objects. References passed between active objects must be unique, either originally or as a result of performing a clone. Note that having `unique` as the default annotation does not apply to active class types appearing in the interface, as these can only be `active`. This choice of defaults is supported by the experimental results of Potanin and Noble [50] and Ma and Foster [39], which show that many arguments between objects could well be unique references.

3.4 Revisiting the example

Figure 4 adds active ownership annotations to Figure 2. As a result, all copying is avoided. Only six annotations are needed to express the intended semantics of

[3] We currently do not support borrowing on asyncronous method calls. A unique object transferred to from active object A to B must be explicitly transferred back.

```
active class Client {
   void run() ‡ {
      †Request rm = ...; // formulate Request
      future immutable Offer offer = myBroker!book(rm); // (1)
      ... // evaluate offer
      offer.getProvider()!accept(offer); // (2)
   }
}
active class Broker {
   void run() ‡ { ... } // go into reactive mode
   // book returns first provider that responds to the request
   immutable Offer book(safe Request request) ‡ { ... } // (3)
}
active class Provider {
   immutable Offer query(safe Request request) ‡ { ... } // (4)
   boolean accept(immutable Offer offer) ‡ { ... } // (5)
}
class Request {
   Request(†String desc) ‡ { ... }
   void markAccepted() ‡ { ... }
}
class Offer {
   Offer(†Details d, †Provider p, safe Request r) ‡ ... // (6)
   Provider getProvider() read { ... } // (7)
}
```

Figure 4. The active objects example with active ownership. † indicates an implicit use of a default, `owner` in passive classes and `this` in active. ‡ indicates an implicit use of the `write` default for methods. These are not part of the actual code.

Figure 2. This might seem excessive for a 20-line program, but remember that we only focus on the parts of the program that needs annotations. Furthermore, no annotations are required for library code used by this program. But more importantly, we can now captures the programmer's intentions in statically checkable annotations.

The offer is made immutable (4), which allows it to be safely shared between concurrently executing clients, brokers and providers. The immutability requirement propagates to the type of the future variable (1) and formal parameter (5). The request is received as a safe reference (3), so the broker may only access its immutable parts which precludes both races and observational exposure. This constraint is reasonable, since changing crucial parts of a request under foot might lead to invalid offers. Parts of the request can still be updated by the client (but not by the broker or any provider), e.g., to store a handle to the accepted offer in it. The safe annotation propagates to (6). Read-only methods are annotated `read` (7). Reading the provider from an immutable offer (2) returns a reference to an active object, which is safe as it does not share any state with the outside world.

3.5 Other Relevant Features

For space reasons, we omit a discussion of exceptions, which is a straightforward addition (they would be immutable), and a discussion on how to deal with globals (see [22]) and focus on the issue of owner-polymorphic methods in the presence of asynchronous method calls.

Owner-polymorphic methods (and their problems) The previous discussion ignored owner-polymorphic methods. An owner-polymorphic method of an active object enables the active object to borrow passive objects from another active object, with the guarantee that it will not keep any references to the borrowed objects. Such methods require care, as they are problematic in the presence of asynchronous method calls. It is easy to see that an asynchronous call could easily lead to a situation where two active objects (`Client` and `Broker`) have access to the same passive objects (`Request`):

1. `Client` lends `Request` to `Broker` via an asynchronous method call.
2. `Client` continues executing on `Request`.
3. `Broker` concurrently operates on `Request`.

We choose the simplest solution to avoid this problem by banning asynchronous calls to owner-polymorphic methods. Alternative approaches would require preventing `Client` from accessing `Request`—or more precisely, objects with the same owner as `Request`—until the method call to `Broker` returned.

4 Related Work

4.1 Ownership Types

Several approaches using ownership types for concurrency control have been proposed. [7, 11] introduce thread-local owners to avoid data races and deadlocks. Guava [7] is presented as an informal collection of rules which would require a significantly more complex ownership types system than the one we present here. Boyapati et al.'s PRFJ [11] encodes a lock ordering in class headers. The rigid structure imposed by this scheme along with explicit threads makes program evolution tedious and the system complex. Another related approach uses Universes instead of ownership types for race safety [24]. In each case the underlying concurrency model is threads and not active objects.

In X10 *place types* statically describe where data resides improving data locality [52], mainly for performance reasons. X10 also sports futures and a shared space of immutable data. Remote data is remotely accessed, and there are no unique references. However, due to the hierarchical memory model of X10, similar to a distributed system, pointer transfer might not be as effective as in a shared memory system.

StreamFlex [56] and its successor Flexotasks [6] are close in spirit to Joëlle. They use a minimal notion of ownership, with little need for annotations, to

handle garbage collection issues in a real-time setting. Objects may be passed between concurrent components without copying (Singularity OS [32] allows this too). StreamFlex's notion of immutability is however more limited than ours and safe or unique references are not supported. In conclusion, the additional limitations of the stream programming approach (compared to our general-purpose approach) allows StreamFlex to use even less syntactic baggage than Joëlle.

4.2 Actors and Active objects

Erlang [5] is relevant as a purely functional language with immutable data. This is a bad fit for OO and encoding of data structures that rely on sharing or object identifiers is difficult or impossible. Carlsson et al. [16] present an under-approximate *message analysis* to detect when messages can be safely shared rather than copied for a mini-Erlang system with cubic worst-case time complexity. Our type-system based approach should fare better for the price of a few additional concepts and annotations.

Symbian OS [43] and the ProActive Java framework [8] use active objects for concurrency. Based on Eiffel, Eiffel// [17] is an active objects system with asynchronous method calls with futures, and SCOOP [41] uses preconditions for task scheduling. In SCOOP, active object boundaries are captured by `separate` annotations on variables, which, in contrast to our proposal, are not statically enforced. (This is partially improved by Nienaltowski [46].) In the original SCOOP proposal, method arguments across active objects have deep copying semantics, with the aforementioned associated pains. Object migration through uniqueness is not supported. Later versions of SCOOP [46] integrate an eager locking mechanism to enable pass-by-reference arguments for non-value objects. An integration of our approach with SCOOP seems fairly straightforward.

CoBoxes [53] impose a hierarchy structure on active objects to control the access to groups of objects. Our proposal permits only a flat collection of active objects. On the other hand, our system allows the transfer of objects between active objects and the sharing of immutable and safe objects.

Different components or active objects communicate data by cloning in, e.g., the coordination language ToolBus [26] and in ASP [18]. In a distributed setting this is vindicated, but, as ToolBus developers [26] observe, copying data is a major source of performance problems. This is exactly the problem our approach aims to address, without introducing data-races, in a statically checkable fashion.

4.3 Software Transactional Memory

Software Transactional Memory is a recent approach to avoiding data races [54]. Atomic blocks are executed optimistically without locking and versioning detects conflicting updates. STM could be used under the hood to implement Emerald's mutually exclusive object regions [33,51]. Ongoing work by Kotselidis et al. [36] adds software transactional memory to ProActive. Preliminary results are promising, but the system retains ProActive's deep-copying semantics even for inter-node computations.

5 Concluding Remarks

We have applied ownership types to a data sharing problem of active objects. Our solution involves a combination of ownership, external uniqueness, and immutability. Our minimal ownership system was defined so that default annotations can be chosen to give a low syntactic overhead. We expect few necessary changes to code for passive objects if our system was implemented in ProActive.

Our system is close in spirit to several existing systems that lack our static checking for things like immutability. No existing active objects system is powerful enough to use minimal safe cloning the way we outlined in Section 3.3.

A prototype compiler for Joëlle is available from the authors.

References

1. G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
2. J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, volume 3086 of *LNCS*, pages 1–25. Springer, June 2004.
3. J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, Nov. 2002.
4. P. S. Almeida. Balloon Types: Controlling sharing of state in data types. In *ECOOP*, volume 1241 of *LNCS*, pages 32–59. Springer, June 1997.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
6. J. Auerbach, D. Bacon, R. Guerraoui, J. Spring, and J. Vitek. Flexible task graphs: A unified restricted thread programming model for java. In *LCTES*, 2008.
7. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, 2000.
8. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer-Verlag, January 2006.
9. H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in ACMSIGPLAN Notices, page 11, Aug. 1977.
10. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.
11. C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
12. C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *POPL*, 2003.
13. C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.
14. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.
15. J. Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, June 2006. Special issue: ECOOP 2005 Workshop FTfJP.
16. R. Carlsson, K. F. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM TOPLAS*, 28(4):715–746, 2006.

17. D. Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Orientated Programming (JOOP)*, pages 12–22, Nov. 1989.
18. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2005.
19. D. Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.
20. D. Clarke and S. Drossopolou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, 2002.
21. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference on Object-Oriented Programming*, volume 2473 of *LNCS*, pages 176–200. Springer, July 2003.
22. D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal Ownership for Active Objects. Technical Report SEN-R0803, CWI, June 2008. `http://ftp.cwi.nl/CWIreports/SEN/SEN-R0803.pdf`.
23. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
24. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, Sept. 2007.
25. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
26. H. de Jong. *Flexible Heterogeneous Software Systems*. PhD thesis, Faculty of Natural Sciences, Math., and Computer Science, Uni. of Amsterdam, Jan. 2007.
27. S. Drossopoulou, D. Clarke, and J. Noble. Types for hierarchic shapes. In P. Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 1–6, 2006.
28. M. Fahndrich and S. Xia. Establishing object invariants with delayed types. *SIGPLAN Not.*, 42(10):337–350, 2007.
29. D. Grossman, M. Hicks, T. Jim, , and G. Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal*, 23(1), Jan. 2005.
30. J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA*, Nov. 1991.
31. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
32. G. Hunt and J. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 40(2):37–49, Apr. 2007.
33. N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul. The Emerald programming language report. Technical Report 87-10-07, Seattle, WA (USA), 1987. Revised 1997.
34. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
35. S. P. Jones and J. H. (editors). Haskell 98: A non-strict, purely functional language. Technical report, Feb. 1999.
36. C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Investigating software transactional memory on clusters. In *IWJPDC '08*. IEEE Computer Society Press, April 2008.
37. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.
38. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 2nd edition, 2000.
39. K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.
40. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

41. B. Meyer. Systematic concurrent object-oriented programming. *CACM*, 36(9):56–80, 1993.
42. N. H. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP*, volume 1098 of *LNCS*, pages 189–209. Springer, July 1996.
43. B. Morris. CActive and Friends. Symbian Developer Network, November 2007. `http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf`.
44. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
45. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Technical Report 263, Fernuniversität Hagen, 1999.
46. P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.
47. J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *TOOLS Pacific*, Melbourne, Australia, Nov. 1999.
48. J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *ECOOP*, 1998.
49. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *IWACO*, 2007.
50. A. Potanin and J. Noble. Checking ownership and confinement properties. In *Formal Techniques for Java-like Programs*, 2002.
51. R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software: Practice and Experience*, 21(1):91–118, 1991.
52. V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Principles and Practice of Parallel Programming*, 2007.
53. J. Schäfer and A. Poetzsch-Heffter. CoBoxes: Unifying active objects and structured heaps. In *FMOODS'08*, volume 5051 of *LNCS*. Springer, 2008.
54. N. Shavit and D. Touitou. Software transactional memory. In *PODC'95*, pages 204–213. ACM Press, 1995.
55. M. Skoglund and T. Wrigstad. Alias control with read-only references. In *Sixth Conference on Computer Science and Informatics*, Mar. 2002.
56. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. In *OOPSLA*, Oct. 2007.
57. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
58. M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.
59. T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Kista, Stockholm, May 2006.
60. T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 4(6):141–159, May 2007.
61. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In OOPSLA'86. *SIGPLAN Notices*, 21(11):258–268, Nov. 1986.
62. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using Java generics. In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 75–84. ACM Press, 2007.