Actors without Borders: Amnesty for Imprisoned State

Elias Castegren Tobias Wrigstad
Uppsala University, Sweden

In concurrent systems, some form of synchronisation is typically needed to achieve data-race freedom, which is important for correctness and safety. In actor-based systems, messages are exchanged concurrently but executed sequentially by the receiving actor. By relying on isolation and non-sharing, an actor can access its own state without fear of data-races, and the internal behavior of an actor can be reasoned about sequentially.

However, actor isolation is sometimes too strong to express useful patterns. For example, letting the iterator of a data-collection alias the internal structure of the collection allows a more efficient implementation than if each access requires going through the interface of the collection. With full isolation, in order to maintain sequential reasoning the iterator must be made part of the collection, which bloats the interface of the collection and means that a client must have access to the whole data-collection in order to use the iterator.

In this paper, we propose a programming language construct that enables a relaxation of isolation but without sacrificing sequential reasoning. We formalise the mechanism in a simple lambda calculus with actors and passive objects, and show how an actor may leak parts of its internal state while ensuring that any interaction with this data is still synchronised.

1 Introduction

Synchronisation is a key aspect of concurrent programs and different concurrency models handle synchronisation differently. Pessimistic models, like locks or the actor model [1] serialise computation *within certain encapsulated units*, allowing sequential reasoning about internal behavior.

In the case of the actor model, for brevity including also active objects (which carry state, which actor's traditionally do not), if a reference to an actor A's internal state is accessible outside of A, operations inside of A are subject to data-races and sequential reasoning is lost. The same holds true for operations on an aggregate object behind a lock, if a subobject is leaked and becomes accessible where the appropriate lock is not held.

In previous work, we designed Kappa [4], a type system in which the boundary of a unit of encapsulation can be statically identified. An entire encapsulated unit can be wrapped inside some synchronisation mechanism, *e.g.,* a lock or an asynchronous actor interface, and consequently all operations inside the boundary are guaranteed to be data-race free. An important goal of this work is facilitating object-oriented reuse in concurrent programming: internal objects are oblivious to how their data-race freedom is guaranteed, and the building blocks can be reused without change regardless of their external synchronisation.

This extended abstract explores two extensions to this system, which we explain in the context of the actor model (although they are equally applicable to a system using locks). Rather than rejecting programs where actors leak internal objects, we allow an actor to *bestow* its synchronisation mechanism upon the exposed objects. This allows multiple objects to effectively construct an actor's interface. Exposing internal operations externally makes concurrency more fine-grained. To allow external control of the possible interleaving of these operations, we introduce an *atomic block* that groups them together. The following section motivates these extensions.

Submitted to:
PLACES 2017

```
class Node[t]
  var next : Node[t]
  var elem : t
  // getters and setters omitted

actor List[t]
  var first : Node[t]
  def getFirst() : Node[t]
    return this.first

  def get(i : int) : t
    var current = this.first
    while i > 0 do
      current = current.next
      i = i - 1
    return current.elem
```

(a)

```
class Iterator[t]
  var current : Node[t]
  def init(first : Node[t]) : void
    this.current = first

  def getNext() : t
    val elem = this.current.elem
    this.current = this.current.next
    return elem

  def hasNext() : bool
    return this.current != null

actor List[t]
  def getIterator() : Iterator[t]
    val iter = new Iterator[t]
    iter.init(this.first)
    return iter
```

(b)

Figure 1: (a) A list implemented as an actor. (b) An iterator for that list.

2 Breaking Isolation: Motivating Example

We motivate breaking isolation in the context of an object-oriented actor language, with actors serving as the units of encapsulation, encapsulating zero or more passive objects. Figure 1a shows a Kappa program with a linked list in the style of an actor with an asynchronous external interface. For simplicity we allow asynchronous calls to return values and omit the details of how this is accomplished (*e.g.,* by using futures, promises, or by passing continuations).

Clients can interact with the list for example by sending the message `get` with a specified index. With this implementation, each time `get` is called, the corresponding element is calculated from the head of the list, giving linear time complexity for each access. Iterating over all the elements of the list has quadratic time complexity.

To allow more efficient element access, the list can provide an iterator which holds a pointer to the current node (Figure 1b). This allows constant-time access to the *current* element, and linear iteration, but also breaks encapsulation by providing direct access to nodes and elements without going through the list interface. *List operations are now subject to data-races.*

A middle ground providing linear time iteration without data-races can be implemented by moving the iterator logic into the list actor, so that the calls to `getNext` and `hasNext` are synchronised in the message queue of the actor. This requires a more advanced scheme to map different clients to different concurrent iterators, clutters the list interface, creates unnecessary coupling between `List` and `Iterator`, and complicates support of *e.g.,* several kinds of iterators.

Another issue with concurrent programs is that interleaving interaction with an actor makes it hard to reason about operations that are built up from several smaller operations. For example, a client might want to access two adjacent nodes in the list and combine their elements somehow. When sending two `get` messages, there is nothing that prevents other messages from being processed by the list actor after the first one, possibly removing or changing one of the values.

```
actor List[t]
  ...
  def getIterator() : B(Iterator[t])
    val iter = new Iterator[t]
    iter.init(this.first)
    return bestow iter
```

```
val iter = list!getIterator()
while iter!hasNext() do
  val elem = iter!getNext()
  ...
```

Figure 2: A list actor returning a bestowed iterator, and the code for a client using it

Again, unless the list actor explicitly provides an operation for getting adjacent values, there is no way for a client to safely express this operation.

3 Bestowing and Grouping Activity

Encapsulating state behind a synchronisation mechanism allows reasoning sequentially about operations on that state. However, since Kappa lets us identify the encapsulation boundary of the data structure [4], it is possible to *bestow* objects that are leaked across this boundary with a synchronisation wrapper. Statically, this means changing the type of the returned reference to reflect that operations on it may block. Dynamically it means identifying with what and how the leaked object shall synchronise.

For clarity, we explicate this pattern with a **bestow** operation. In the case of actors, an actor `a` that performs **bestow** on some reference r creates a wrapper around r that makes it appear like an actor with the same interface as r, *but asynchronous*. Operations on the bestowed reference will be relayed to `a` so that the actor `a` is the one actually performing the operation. If r was leaked from an enclosure protected by a lock `l`, r's wrapper would instead acquire and release `l` around each operation.

Figure 2 shows the minimal changes needed to the code in Figure 1b, as well as the code for a client using the iterator. The only change to the list is that `getIterator()` returns a bestowed iterator (denoted by wrapping the return type in **B**`(...)`[1]), rather than a passive one. In the client code, synchronous calls to `hasNext()` and `getNext()` become asynchronous message sends. These messages are handled by the list actor, even though they are not part of its interface. This means that any concurrent usages of iterators are still free from data-races.

It is interesting to ponder the difference between creating an iterator *inside* the list and bestowing it, or creating an iterator *outside* the list, and bestowing each individual list node it traverses. In the former case, `getNext()` is performed without interleaved activities in the same actor. In the latter case, it is possible that the internal operations are interleaved with other operations on list. The smaller the object returned, the more fine-grained is the concurrency.

Sometimes it is desirable that multiple operations on an object are carried out in a non-interleaved fashion. For this purpose, we use an **atomic** block construct that operates on a an actor or a bestowed object, *cf.* Figure 3. In the case of operations on an actor, message sends inside an atomic block are *batched* and sent as a single message to the receiver. In the case of operations on an object guarded by a lock, we replace each individual lock–release by a single lock–release wrapping the block. It is possible to synchronise across multiple locked objects in a single block.

[1] If desired, this type change can be implicit through view-point adaptation [9].

```
class Iterator[t]
  var current : B(Node[t])
  def getNext() : t
    val elem = this.current ! elem()
    // Possible interleaving of other messages
    this.current = this.current ! next()
    return elem
```

```
class Iterator[t]
  var current : B(Node[t])
  def getNext() : t
    atomic c <- this.current
      val elem = c ! elem()
      this.current = c ! next()
      return elem
```

Figure 3: Fine-grained (left) and coarse-grained (right) concurrency control.

An **atomic** block allows a client to express new operations by composing smaller ones. The situation sketched in § 2, where a client wants to access two adjacent nodes in the list actor without interleaving operations from other clients is easily resolved by wrapping the two calls to `get` (or `getNext`, if the iterator is used) inside an **atomic** block. This will batch the messages and ensure that they are processed back to back:

```
atomic it <- list ! getIterator()
  val e1 <- it.getNext()
  val e2 <- it.getNext()
```

$\Longrightarrow$

```
(e1, e2) =
  list ! λ this .
    {val it = this.getIterator();
     val e1 = it.getNext();
     val e2 = it.getNext();
     return (e1, e2)}
```

4 Formalism

To explain **bestow** and **atomic** we use a simple lambda calculus with actors and passive objects. We abstract away most details that are unimportant when describing the behavior of bestowed objects. For example, we leave out classes and actor interfaces and simply allow arbitrary operations on values. By disallowing sharing of (non-bestowed) passive objects, we show that our language is free from data-races (*cf.* § 4.4).

The syntax of our calculus is shown in Figure 4. An expression e is a variable x, a function application $e\ e'$ or a message send $e!v$. Messages are sent as anonymous functions, which are executed by the receiving actor. We abstract updates to passive objects as $e.\texttt{mutate}()$, which has no actual effect in the formalism, but is reasoned about in § 4.4. A new object or actor is created with **new** τ and a passive object can be bestowed by the current actor with **bestow** e. We don't need a special **atomic** construct in the formalism as this can be modeled by composing operations in a single message as sketched in the end of the previous section.

Statically, values are anonymous functions or the unit value (). Dynamically, id is the identifier of an actor, ι is the memory location of a passive object, and ι_{id} is a passive object ι bestowed by the actor id. A type is an active type α, a passive type p, a function type $\tau \to \tau$, or the `Unit` type. An active type is either an actor type c or a bestowed type $\mathbf{B}(\mathsf{p})$. Note that for simplicity,

$$
\begin{array}{lcl}
e & ::= & x \mid e\ e \mid e!v \mid e.\texttt{mutate}() \mid \textbf{new}\ \tau \mid \textbf{bestow}\ e \mid v \\
v & ::= & \lambda x : \tau . e \mid () \mid id \mid \iota \mid \iota_{id}
\end{array}
\qquad
\begin{array}{lcl}
\tau & ::= & \alpha \mid \mathsf{p} \mid \tau \to \tau \mid \texttt{Unit} \\
\alpha & ::= & \mathsf{c} \mid \mathbf{B}(\mathsf{p})
\end{array}
$$

Figure 4: The syntax of a simple lambda calculus with actors, **bestow** and **atomic**.

p and c are not meta-syntactic variables; every passive object has type p, every actor has type c, and every bestowed object has type $\mathbf{B}(\mathsf{p})$.

$\boxed{\Gamma \vdash e : \tau}$ *(Expressions)*

$$\textsc{e-var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{e-apply}\ \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\, e' : \tau} \qquad \textsc{e-new-passive}\ \frac{}{\Gamma \vdash \mathbf{new}\,\mathsf{p} : \mathsf{p}} \qquad \textsc{e-new-actor}\ \frac{}{\Gamma \vdash \mathbf{new}\,\mathsf{c} : \mathsf{c}}$$

$$\textsc{e-mutate}\ \frac{\Gamma \vdash e : \mathsf{p}}{\Gamma \vdash e.\texttt{mutate}() : \texttt{Unit}} \qquad \textsc{e-bestow}\ \frac{\Gamma \vdash e : \mathsf{p}}{\Gamma \vdash \mathbf{bestow}\, e : \mathbf{B}(p)} \qquad \textsc{e-send}\ \frac{\Gamma \vdash e : \alpha \quad \Gamma_\alpha, x : \mathsf{p} \vdash e' : \tau' \quad \nexists\, \iota\ .\ \iota \in e'}{\Gamma \vdash e\,!\,\lambda x : \mathsf{p}.e' : \texttt{Unit}}$$

$$\textsc{e-fn}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.e) : \tau \to \tau'} \qquad \textsc{e-unit}\ \frac{}{\Gamma \vdash () : \texttt{Unit}} \qquad \textsc{e-loc}\ \frac{}{\Gamma \vdash \iota : \mathsf{p}} \qquad \textsc{e-id}\ \frac{}{\Gamma \vdash id : \mathsf{c}} \qquad \textsc{e-bestowed}\ \frac{}{\Gamma \vdash \iota_{id} : \mathbf{B}(p)}$$

Figure 5: Static semantics. Γ maps variables to types. Γ_α contains only the active types α of Γ.

4.1 Static Semantics

The typing rules for our formal language can be found in Figure 5. The typing context Γ maps variables to types. The "normal" lambda calculus rules E-VAR and E-APPLY are straightforward. The **new** keyword can create new passive objects or actors (E-NEW-*). Passive objects may be mutated (E-MUTATE), and may be bestowed activity (E-BESTOW).

Message sends are modeled by sending anonymous functions which are run by the receiver (E-SEND). The receiver must be of active type (*i.e.,* be an actor or a bestowed object), and the argument of the anonymous function must be of passive type p (this can be thought of as the **this** of the receiver). Finally, all free variables in the body of the message must have active type to make sure that passive objects are not leaked from their owning actors. This is captured by Γ_α which contains only the active mappings $\_ : \alpha$ of Γ. Dynamically, the body may not contain passive objects ι. Typing values is straightforward.

4.2 Dynamic Semantics

Figure 6 shows the small-step operational semantics for our language. A running program is a heap H, which maps actor identifiers id to actors (ι, L, Q, e), where ι is the **this** of the actor, L is the local heap of the actor (a set containing the passive objects created by the actor), Q is the message queue (a list of lambdas to be run), and e is the current expression being evaluated.

An actor whose current expression is a value may pop a message from its message queue and apply it to its **this** (EVAL-ACTOR-MSG). Any actor in H may step its current expression, possibly also causing some effect on the heap (EVAL-ACTOR-RUN). The relation $id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle$ denotes actor id evaluating heap H and expression e one step.

$\boxed{H \hookrightarrow H'}$ *(Evaluation)*

$$\textsc{eval-actor-msg}\ \frac{H(id) = (\iota, L, Q\,v', v) \qquad H' = H[id \mapsto (\iota, L, Q, v'\,\iota)]}{H \hookrightarrow H'}$$

$$\textsc{eval-actor-run}\ \frac{H(id) = (\iota, L, Q, e) \qquad id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle \qquad H'(id) = (\iota, L', Q', e) \qquad H'' = H'[id \mapsto (\iota, L', Q', e')]}{H \hookrightarrow H''}$$

$\boxed{id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle}$ *(Evaluation of expressions)*

$$\textsc{eval-send-actor}\ \frac{H(id') = (\iota, L, Q, e) \qquad H' = H[id' \mapsto (\iota, L, v\,Q, e)]}{id \vdash \langle H, id' \,!\, v\rangle \hookrightarrow \langle H', ()\rangle}$$

$$\textsc{eval-send-bestowed}\ \frac{H(id') = (\iota', L, Q, e) \qquad H' = H[id' \mapsto (\iota', L, (\lambda x : \mathsf{p}.v\,\iota)\,Q, e)]}{id \vdash \langle H, \iota_{id'} \,!\, v\rangle \hookrightarrow \langle H', ()\rangle}$$

$$\textsc{eval-apply}\ \frac{e' = e[x \mapsto v]}{id \vdash \langle H, (\lambda x : \tau.e)\,v\rangle \hookrightarrow \langle H, e'\rangle} \qquad \textsc{eval-mutate}\ \frac{}{id \vdash \langle H, \iota.\texttt{mutate}()\rangle \hookrightarrow \langle H, ()\rangle} \qquad \textsc{eval-bestow}\ \frac{}{id \vdash \langle H, \mathbf{bestow}\,\iota\rangle \hookrightarrow \langle H, \iota_{id}\rangle}$$

$$\textsc{eval-new-passive}\ \frac{H(id) = (\iota, L, Q, e) \qquad \iota'\,\mathbf{fresh} \qquad H' = H[id \mapsto (\iota, L \cup \{\iota'\}, Q, e)]}{id \vdash \langle H, \mathbf{new}\,\mathsf{p}\rangle \hookrightarrow \langle H', \iota'\rangle} \qquad \textsc{eval-new-actor}\ \frac{id'\,\mathbf{fresh} \qquad \iota'\,\mathbf{fresh} \qquad H' = H[id' \mapsto (\iota', \{\iota'\}, \epsilon, ())]}{id \vdash \langle H, \mathbf{new}\,\alpha\rangle \hookrightarrow \langle H', id'\rangle}$$

$$\textsc{eval-context}\ \frac{id \vdash \langle H, e\rangle \hookrightarrow \langle H', e'\rangle}{id \vdash \langle H, E[e]\rangle \hookrightarrow \langle H', E[e']\rangle}$$

$E[\bullet] ::= \bullet\ e \mid v\ \bullet \mid \bullet\,!v \mid \bullet.\text{mutate}() \mid \mathbf{bestow}\ \bullet$

Figure 6: Dynamic semantics.

Sending a lambda to an actor prepends this lambda to the receiver's message queue and results in the unit value (EVAL-SEND-ACTOR). Sending a lambda v to a bestowed value instead prepends a new lambda to the queue of the actor that bestowed it, which simply applies v to the underlying passive object (EVAL-SEND-BESTOWED).

Function application replaces all occurrences of the parameter x in its body by the argument v (EVAL-APPLY). Mutation is a no-op in practice (EVAL-MUTATE). Bestowing a passive value ι in actor id creates the bestowed value ι_{id} (EVAL-BESTOW).

Creating a new object in actor id adds a fresh location ι' to the set of the actors passive objects L and results in this value (EVAL-NEW-PASSIVE). Creating a new actor adds a new actor with a fresh identifier to the heap. Its local heap contains only the fresh **this**, its queue is empty, and its current expression is the unit value (EVAL-NEW-ACTOR).

We handle evaluation order by using an evaluation context E (EVAL-CONTEXT).

4.3 Well-formedness

A heap H is well-formed if all its actors are well-formed with respect to H, and the local heaps L_i and L_j of any two different actors are disjoint (WF-HEAP). We use $\mathcal{LH}(H(id))$ to denote the local heap of actor id. An actor is well-formed if its **this** is in its local heap L and its message

$\boxed{\vdash H \qquad H \vdash (\iota, L, Q, e) \qquad H \vdash Q}$ *(Well-formedness)*

$$\textsc{wf-heap}\ \frac{\forall id_1 \neq id_2\ .\ \mathcal{LH}(H(id_1)) \cap \mathcal{LH}(H(id_2)) = \emptyset \qquad \forall id \in \mathbf{dom}(H)\ .\ H \vdash H(id)}{\vdash H}$$

$$\textsc{wf-actor}\ \frac{\iota \in L \qquad H; L \vdash Q \qquad \epsilon \vdash e : \tau \qquad \forall \iota \in e\ .\ \iota \in L \qquad \forall id \in e\ .\ id \in \mathbf{dom}(H) \qquad \forall \iota_{id} \in e\ .\ \iota \in \mathcal{LH}(H(id))}{H \vdash (\iota, L, Q, e)}$$

$$\textsc{wf-queue-message}\ \frac{H; L \vdash Q \qquad x : \mathsf{p} \vdash e : \tau \qquad \forall \iota \in e\ .\ \iota \in L \qquad \forall id \in e\ .\ id \in \mathbf{dom}(H) \qquad \forall \iota_{id} \in e\ .\ \iota \in \mathcal{LH}(H(id))}{H; L \vdash (\lambda x : \mathsf{p}.e)\ Q}$$

$$\textsc{wf-queue-empty}\ \frac{}{H; L \vdash \epsilon}$$

Figure 7: Well-formedness rules. $\mathcal{LH}$ gets the local heap from an actor: $\mathcal{LH}((\iota, L, Q, e)) = L$

queue Q is well-formed. The current expression e must be typable in the empty environment, and all passive objects ι that are subexpressions of e must be in the local heap L. Similarly, all actor identifiers in e must be actors in the system, and all bestowed objects must belong to the local heap of the actor that bestowed it (WF-ACTOR).

A message queue is well-formed if all its messages are well-formed (WF-QUEUE-*). A message is well-formed if it is a well-formed anonymous function taking a passive argument, and has a body e with the same restrictions on values as the current expression in an actor.

4.4 Meta Theory

We prove soundness of our language by proving progress and preservation in the standard fashion:

> **Progress**: A well-formed heap H can either be evaluated one step, or only has actors with empty message queues and fully reduced expressions:
>
> $$\vdash H \implies (\exists H'\ .\ H \hookrightarrow H')\ \vee\ (\forall id \in \mathbf{dom}(H)\ .\ H(id) = (\iota, L, \epsilon, v))$$
>
> **Preservation**: Evaluation preserves well-formedness of heaps: $\vdash H\ \wedge\ H \hookrightarrow H' \implies \vdash H'$

Both properties can be proven to hold with straightforward induction.

The main property that we are interested in for our language is data-race freedom. As we don't have any actual effects on passive objects, we show this by proving that if an actor is about to execute ι.`mutate()`, no other actor will be about to execute `mutate` on the same object:

> **Data-race freedom**: Two actors will never mutate the same active object
>
> $$\left(\begin{array}{c} id_1 \neq id_2 \\ \wedge\ H(id_1) = (\iota_1, L_1, Q_1, \iota.\texttt{mutate()}) \\ \wedge\ H(id_2) = (\iota_2, L_2, Q_2, \iota'.\texttt{mutate()}) \end{array}\right) \implies \iota \neq \iota'$$

This property is simple to prove using two observations on what makes a well-formed heap:

1. An actor will only ever access passive objects that are in its local heap (WF-ACTOR).
2. The local heaps of all actors are disjoint (WF-HEAP).

The key to showing preservation of the first property is in the premise of rule E-SEND which states that all free variables and values must be active objects ($\Gamma_\alpha, x : \mathsf{p} \vdash e' : \tau'$ and $\nexists \iota \,.\, \iota \in e'$). This prevents sending passive objects between actors without bestowing them first. Sending a message to a bestowed object will always relay it to the actor that owns the underlying passive object (by the premise of WF-ACTOR: $\forall \iota_{id} \in e \,.\, \iota \in \mathcal{LH}(H(id))$). Preservation of the second property is simple to show since local heaps grow monotonically, and are only ever extended with fresh locations (EVAL-NEW-PASSIVE).

Having made these observations, it is trivial to see that an actor in a well-formed heap H that is about to execute ι.`mutate()` must have ι in its own local heap. If another actor is about to execute ι'.`mutate()`, ι' must be in the local heap of this actor. As the local heaps are disjoint, ι and ι' must be different. Since well-formedness of heaps are preserved by evaluation, all programs are free from data-races.

5 Related Work

An important property of many actor-based systems is that a single actor can be reasoned about sequentially; messages are exchanged concurrently but executed sequentially by the receiving actor. For this property to hold, actors often rely on *actor isolation* [10], *i.e.,* that the state of one actor cannot be accessed by another. If this was the not the case, concurrent updates to shared state could lead to data-races, breaking sequential reasoning.

Existing techniques for achieving actor isolation are often based on restricting aliasing, for example copying all data passed between actors [2], or relying on linear types to transfer ownership of data [3, 5, 6, 10]. Bestowed objects offer an alternative technique which relaxes actor isolation and allows sharing of data without sacrificing sequential reasoning. Combining bestowed objects with linear types is straightforwand and allows for both ownership transfer and bestowed sharing between actors in the same system.

Miller *et al.*propose a programming model based on function passing, where rather than passing data between concurrent actors, functions are sent to collections of stationary and immutable data called *silos* [7]. Bestowed objects are related in the sense that sharing them doesn't actually move data between actors. In the function passing model, they could be used to provide an interface to some internal part of a silo, but implicitly relay all functions passed to it to its owning silo. While the formalism in § 4 also works by passing functions around, this is to abstract away from unimportant details, and not a proposed programming model.

References to bestowed objects are close in spirit to remote references in distributed programming or eventual references in E [8]. In the latter case, the unit of encapsulation, *e.g.,* an actor or an aggregate object protected by a lock, acts similar to a Vat in E, but with an identifiable boundary and an identity with an associated interface. By bestowing and exposing sub-objects, a unit of encapsulation can safely delegate parts of its interface to its inner objects, which in turn need not be internally aware of the kind of concurrency control offered by their bestower.

6 Discussion

Although our formal description and all our examples focus on actors, **bestow** also works with threads and locks. An object protected by a lock can share one of its internal objects while requiring that any interaction with this object also goes via this lock. We believe there is also a straightforward extension to software transactional memory. In the future, we would like to study combinations of these.

Bestowed objects lets an actor expose internal details about its implementation. Breaking encapsulation should always be done with care as leaking abstractions leads to increased coupling between modules and can lead to clients observing internal data in an inconsistent state. The latter is not a problem for bestowed objects however; interactions with bestowed objects will be synchronised in the owning actor's message queue, so as long as data is always consistent *between* messages, we can never access data in an inconsistent state (if your data is inconsistent between messages, you have a problem with or without bestowed objects).

Sharing bestowed objects may increase contention on the owner's message queue as messages to a bestowed object are sent to its owner. Similarly, since a bestowed object is protected by the same lock as its owner, sharing bestowed objects may lead to this lock being polled more often. As always when using locks there is a risk of introducing deadlocks, but we do not believe that bestowed objects exacerbate this problem. Deadlocks caused by passing a bestowed object back to its owner can be easily avoided by using reentrant locks (as accessing them both would require taking the same lock twice).

When using locks, **atomic** blocks are very similar to Java's **synchronized**-blocks. With actors, an **atomic** block groups messages into a single message. For fairness, it may make sense to only allow **atomic** blocks that send a limited number of messages.

It is possible to synchronise on several locked objects by simply grabbing several locks. Synchronising on several actors is more involved, as it requires actors to wait for each other and communicate their progress so that no actor starts or finishes before the others. The canonical example of this is atomically withdrawing and depositing the same amount from the accounts of two different actors. Interestingly, if the accounts are bestowed objects from the same actor (*e.g.,* some bank actor), this atomic transaction can be implemented with the message batching approach suggested in this paper. We leave this for future work.

6.1 Implementation

We are currently working on implementing bestowed objects and **atomic** blocks in the context of Encore [3], which uses active objects for concurrency. In Encore, each object (passive or active) has an interface defined by its class, and only the methods defined therein may be invoked. Thus it does not follow the formal model from § 4, where message passing is implemented by sending anonymous functions. It does however use the same approach for the implementation of bestowed objects and **atomic** blocks.

We extend each active class with an implicit method `perform` which takes a function, applies it to the **this** of the receiver, and returns the result wrapped in a future. A bestowed object is logically implemented as an object with two fields `owner` and `object`. A message send `x ! foo()` to a bestowed object is translated into the message send `x.owner ! perform((λ _ . x.object.foo()))`.

The **atomic** block can be implemented as sketched in the end of § 3, where messages are batched and sent as a single message:

```
atomic x <- e
  x ! foo(42)    ⟹  e ! perform(λ this . {this.foo(42); this.bar(-42)})
  x ! bar(-42)
```

This implementation works for the use-cases discussed here, but is somewhat limiting as it doesn't allow the caller to react to intermediate values. We are therefore exploring an alternative approach where we temporarily switch the message queue of an active object to one that only the caller can submit messages to. Other messages passed to the active object will end up in the original message queue, and will be processed first when the **atomic** block finishes.

Each active object would implicitly be extended with two methods `override`, which switches the current message queue to a new one, and `resume`, which discards the temporary queue and resumes execution with the original queue. Logically, the translation could look like this:

```
atomic x <- e                          val q = new MessageQueue()
  val v1 = x ! foo(42)                 e ! override(q) // 1
  val v2 = this.bar(v1)        ⟹      val v1 = q.enqueue(("foo", [42]))
  x ! baz(v2)                          val v2 = this.bar(v1)
                                       q.enqueue(("baz", [v2]))
                                       q.enqueue(("resume", [])) // 2
```

When the message at `1` is processed by receiver, it stops reading from its regular message queue and instead starts using the queue provided by the caller. Rather than sending messages normally, the caller interacts with `x` through this queue (waiting for responses if necessary). When the message at `2` has been processed by the receiver, it goes back to reading messages normally.

6.2 Abstracting Over Synchronisation Methods

Finally, we note the connection to the **safe** type qualifier introduced by the Kappa type system [4], which ranges over both actors and locks (and immutables etc.). A value with a **safe** type can be accessed concurrently without risk of data-races, but how this is achieved depends on the type of the value at runtime. Let `x` have the type **safe** τ. Now, `z = x.foo()` is equivalent to `z = x!foo().get()` when `x` is an actor returning a future value, and `get()` is a blocking read on the future. When `x` is protected by a lock `l`, the same access is equivalent to `lock(l); z = x.foo(); unlock(l);`. When `x` is immutable, no special synchronisation is needed.

Consequently, the **safe** qualifier can be used to express operations on objects with concurrency control abstracted out, without losing safety. An **atomic** block can be used to atomically compose operations on a **safe** object, and the choice of concurrency control mechanism can be relegated to the runtime. Similarly, bestowed objects internally has no knowledge about their own concurrency control. Thus, when a bestowed object is used as a **safe** object, neither the object itself nor its client needs knows how the interaction is made safe.

7 Conclusion

Actor isolation is important to maintain sequential reasoning about actors' behavior. By bestowing activity on its internal objects, an actor can share its representation without losing sequential reasoning and without bloating its own interface. With **atomic** blocks, a client can create new behavior by composing smaller operations. The bestowed objects themselves do not need to know why access to them is safe. They can just trust the safety of living in a world where actors have no borders.

References

[1] G. Agha (1986): *Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence*. *MIT Press* 11.

[2] J. Armstrong (2007): *A History of Erlang*. In: *HOPL III*, doi:10.1145/1238844.1238850.

[3] S Brandauer et al. (2015): *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*. In: *Formal Methods for Multicore Programming*, doi:10.1007/978-3-319-18941-3_1.

[4] E. Castegren & T. Wrigstad (2016): *Reference Capabilities for Concurrency Control*. In: *ECOOP*, doi:10.4230/LIPIcs.ECOOP.2016.5.

[5] S. Clebsch, S. Drossopoulou, S. Blessing & A. McNeil (2015): *Deny Capabilities for Safe, Fast Actors*. In: *AGERE*, doi:10.1145/b2824815.2824816.

[6] P. Haller & M. Odersky (2010): *Capabilities for Uniqueness and Borrowing*. In: *ECOOP*, doi:10.1007/978-3-642-14107-2_17.

[7] Heather Miller, Philipp Haller, Normen Müller & Jocelyn Boullier (2016): *Function Passing: A Model for Typed, Distributed Functional Programming*. In: *Onward!*, doi:10.1145/2986012.2986014.

[8] M. Miller (2006): *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. thesis, Johns Hopkins University, USA.

[9] P. Müller (2002): *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-45651-1.

[10] S. Srinivasan & A. Mycroft (2008): *Kilim: Isolation-Typed Actors for Java*. In: *ECOOP*, doi:10.1007/978-3-540-70592-5_6.