Domains: Safe sharing among actors

Joeri De Koster Tom Van Cutsem Theo D'Hondt
Vrije Universiteit Brussel,
Pleinlaan 2,
B-1050 Brussels, Belgium
jdekoste@vub.ac.be tvcutsem@vub.ac.be tjdhondt@vub.ac.be

Abstract

The actor model has already proven itself as an interesting concurrency model that avoids issues such as deadlocks and race conditions by construction, and thus facilitates concurrent programming. While it has mainly been used in a distributed context it is certainly equally useful for modeling interactive components in a concurrent setting. In component based software, the actor model lends itself to naturally dividing the components over different actors and using message passing concurrency for implementing the interactivity between these components. The tradeoff is that the actor model sacrifices expressiveness and efficiency especially with respect to parallel access to shared state.

This paper gives an overview of the disadvantages of the actor model in the case of shared state and then formulates an extension of the actor model to solve these issues. Our solution proposes *domains* and *synchronization views* to solve the issues without compromising on the semantic properties of the actor model. Thus, the resulting concurrency model maintains deadlock-freedom and avoids low-level race conditions.

1. Introduction

Traditionally, concurrency models fall into two broad categories: message-passing versus shared-state concurrency control. Both models have their relative advantages and disadvantages. In this paper, we explore an extension to a message-passing concurrency model that allows safe, expressive and efficient sharing of mutable state among otherwise isolated concurrent components.

A well-known message-passing concurrency model is the actor model [3]. In this model, applications are decomposed into concurrently running actors. Actors are isolated (i.e., have no direct access to each other's state), but may interact via (asynchronous) message passing. While originally designed to model open, distributed systems, and thus often used as a distributed programming model, they remain equally useful as a more high-level alternative to shared-memory multithreading. Both component-based and service-oriented architectures can be modeled naturally using actors. It is important to point out that in this paper, we restrict ourselves to the use of actors as a concurrency model, not as a distribution model.

AGERE! @ SPLASH '12 21-22 October 2012 - Tucson, Arizona, USA.

In practice, the actor model is either made available via dedicated programming languages (actor languages), or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they often strictly enforce the isolation of actors: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. Examples of pure actor languages include Erlang [5], E [18], AmbientTalk [23], Salsa [24] and Kilim [20]. The major benefit of pure actor languages is that the developer gets very strong safety guarantees: low-level race conditions are avoided. On the other hand, these languages make it difficult to express shared mutable state. Often, one needs to express shared state in terms of a shared actor encapsulating that state, which has several disadvantages, as will be discussed in Section 2.4.

On the other end of the spectrum, we find actor libraries, which are very often added to an existing language whose concurrency model is based on shared-memory multithreading. For Java alone, there exist the ActorFoundry [6], AsyncObjects [2], ... Scala, which inherits shared-memory multithreading from Java, features multiple actor frameworks, such as Scala Actors [12] and Akka [1]. What these libraries have in common is that they cannot typically enforce actor isolation, i.e. they do not guarantee that actors don't share mutable state. On the other hand, it's easy for a developer to use the underlying shared-memory concurrency model as an "escape hatch" when direct sharing of state is the most natural or most efficient solution. However, once the developer chooses to go this route, all of the benefits of the high-level actor model are lost, and the developer typically has to resort to manual locking to prevent data races.

The goal of this work is to enable safe, expressive and efficient state sharing among actors:

safe : the isolation properties of actors are often helpful to bring structure to, and help reason about, large-scale software. Consider for instance a plug-in or component architecture. By running plug-ins in their own isolated actors, we can guarantee that they do not violate invariants of the "core" application. Thus, as in pure actor languages, we want an actor system that maintains strong language-enforced guarantees, such as the fact that low-level data races and deadlocks are prevented by design.

expressive : many phenomena in the real world can be naturally modelled using message-passing concurrency (e.g. telephone calls, e-mail, digital circuits, discrete-event simulations, etc.). Sometimes, however, a phenomenon can be modelled more directly in terms of shared state. Consider for instance the scoreboard in a game of football, which can be read in parallel by thousands of spectators. As in impure actor libraries, we want an actor system in which one can directly express access to shared mutable state, without having to encode shared state via a shared actor. Furthermore, by enabling direct *synchronous* access to shared state, we gain stronger synchronization con-

straints and prevent the inversion of control that is characteristic of interacting with actors (as interaction is typically asynchronous).

efficient : today, multicore hardware is becoming the prevalent computing platform, both on the client and the server [21]. While multiple isolated actors can be perfectly executed in parallel by different hardware threads, shared access to a single actor can still form a serious sequential bottleneck. In particular, in pure actor languages that support mutable state, all requests sent to an actor are typically serialized, even if some requests could be processed in parallel (e.g. requests to simply read or query some of the actor's state). Pure actors lack multiple-reader, single-writer access, which is required to enable truly parallel reads of shared state.

In this paper, we propose *domains*, an extension to the actor model that enables safe, expressive and efficient sharing among actors. Since we want to provide strong language-level guarantees, we present domains as part of a small actor language called *Shacl*[1]. In terms of sharing state, our approach strikes a middle ground between what is achievable in a pure actor language versus what can be achieved using impure actor libraries. An interpreter for the whole SHACL language can be found on our website[2].

In the next section 2 we present a number of problems that occur when representing shared state within the actor model. In section 3 we present our domain and view abstractions. In section 4 we list a number of important additional features of SHACL. And to conclude the paper we have a related work section and finally a conclusion.

2. The problem: Accessing non-local shared state

In this section we introduce SHACL, a small implementation of a pure event-loop actor language for which we introduce new features to allow synchronous access to shared state as we go along. There are two ways to represent shared state in the event-loop actor model: either by replicating the shared state over the different actors or by encapsulating the shared state as an additional independent actor. In this section we discuss the disadvantages of both approaches using a motivating example.

2.1 Shacl: An event-loop actor language

The sequential subset of SHACL implements a prototype-based object model similar to Self [22]. This object model also has an inheritance model, supports late binding and static super references. However, these are not relevant in the context of this paper and thus will not be discussed. The concurrency model of SHACL is based on the event-loop model of E [18] and AmbientTalk [23] where actors are represented by *vats*. Each vat/actor has a single thread of execution, an object heap, and an event queue. Each object in the object heap of an actor is *owned* by that actor."Owning" an object gives that actor exclusive access rights to that object. Any reference to an object owned by the same actor is called a *local reference*. A reference to an object owned by another actor is called a *remote reference*. The type of reference determines the access capabilities of the actor on the referenced object. While objects pointed to by local references can be synchronously accessed, any remote object can only be accessed through asynchronous message passing. Thus, sending a message to another actor in this model is just a matter of sending a message to a remote object in that actor's object heap. Any incoming message is then added to the event queue of the actor that owns the remote object. The thread of execution of that actor combined with the event queue form the event-loop of that actor. This event-loop processes arriving messages one by one. The processing of a single message is called a *turn*. Each of those turns is processed in a single atomic step. As in E and AmbientTalk, SHACL supports non-blocking futures to implement two-way messages without using callbacks (See section 4.2). This model was extended with the notion of *domains* to allow shared state between different actors in a controlled way.

2.2 Motivating example

As a motivating example we looked into plugin architectures. Isolation and encapsulation are important properties for such architectures to model the different plugins. Hence, the event-loop actor model is a good fit for such application as it already enforces these properties on the level of the language. The actor model lends itself to naturally model the different plugins each as an actor and using message passing concurrency to model the communication between these different components of the application. However, in such applications, it is common for different plugins to require access to a shared resource. Whether it be globally available for all plugins or just shared between a subset of the plugins.

Figure 1 shows a model of an application where two different plugins use a binary search tree (BST) as a shared data-structure. The binary search tree can be queried for a certain key using `query` and users can insert key-value pairs using `insert`. In our application Plugin 1 will periodically insert new key-value pairs in the tree and Plugin 2 will first query the tree and depending on the result insert a new key-value pair in the tree.

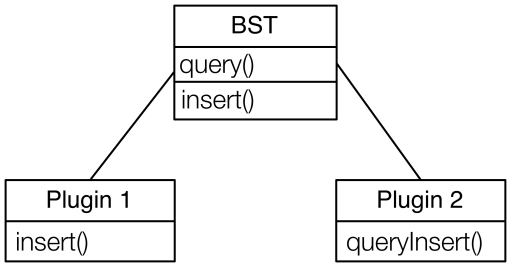

Figure 1. Two plugins using the same shared resource

In the next two sections we will use this example to motivate the issues of using shared state within the actor model.

2.3 Replication

One option for representing shared state in the actor model would be to replicate this state inside different actors that require access to it. For our specific example this would mean that the BST would be replicated in both plugin 1 and 2. This approach has a number of issues:

Consistency Keeping replicated state consistent requires a consistency protocol that usually does not scale well with the number of participants. In our specific example this approach can be used but if we consider applications with hundreds of components this no longer feasible. Lowering the availability or consistency of shared state can be a solution to this problem [9]. Unfortunately, whether lowering either availability or consistency is possible is entirely application dependent. In general, keeping replicated state consistent is a hard problem that usually leads to inefficient code.

Memory usage increases linearly with the amount of shared state and the number of actors. Depending on the granularity with which actors are created, this might incur a memory overhead that is too high.

[1] Pronounce as "shackle", short for **sh**ared **ac**tor **l**anguage

[2] http://soft.vub.ac.be/~jdekoste/shacl

Copying cost Sometimes short lived objects need to be shared between different actors and the cost of copying them is greater than the cost of the operation that needs to be performed on them.

For our approach, we could try to hide these consistency protocols and try to lower the memory usage and copying costs. Unfortunately this solution does not scale very well with the number of actors, making it very unfeasible to use.

2.4 Shared state as an additional independent actor

Using a separate actor to encapsulate shared state is the more natural solution as it does not require any consistency protocols and also scales well with the number of other actors accessing that shared state. There are however three different classes of problems when using this approach:

Continuation-passing style enforced. Using a distinct actor to represent conceptually shared state implies that this resource can not be accessed directly from any other actor since all communication happens asynchronously within the actor model. Thus, the programmer needs to explicitly handle a request-response situation, which usually forces the programmer to employ an explicit continuation-passing-style.

No synchronization conditions. The traditional actor model does not allow specifying extra synchronization conditions on different operations since the order in which events from different senders are handled is nondeterministic.

No parallel reads. State that is conceptually shared can never be read truly in parallel because all accesses to this state are sequentialized by the event queue of the encapsulating actor.

Figure 2 shows an implementation of the BST that is encapsulated by a separate actor. Similarly to E [18] and AmbientTalk [23] in SHACL the `actor{<expression>}` syntax evaluates to a new actor with its own separate object heap and event-loop. That object heap is then initialized with a single new object initialized by `<expression>`. The actor syntax immediately evaluates to a remote reference to that newly created object. Any asynchronous message that is sent to that reference will then be scheduled as an event in the event-loop of the newly created actor. For this example we intentionally left out the implementation details of the BST. What is important here is its interface and how it can be accessed. In this example, querying or updating the BST requires the use of asynchronous communication. Because of that, querying the BST for a value requires the use of callbacks to implement the response message. In our example this is done by passing a callback object, namely the `client` (lines 19–25), as a second argument of the query method. This callback object then has to implement a queried method that is called with the result of the query method.

In this example the `insert` method of plugin 1 just delegates any insert calls to the BST. On the other hand, plugin 2 provides a `queryInsert` method that will query the BST for a certain key and will then decrement the value of that key if it is positive.

In this section we will discuss the three issues raised above in more detail using our motivating example.

Asynchronous communication leads to continuation passing style

The style of programming where a computation is divided into different execution steps is called continuation passing style (CPS), also known as programming without a call stack [15]. This problem of using CPS to access a remote resource is typical and can be found in various other actor languages like Salsa [24], Kilim [20], etc. The problem with this style of programming is that it leads to "inversion of control".

```
1  let bst = actor {
2    insert(key, value) {
3      ...
4    }
5    query(key, client) {
6      result := ...
7      client<-queried(result);
8    }
9  }
10
11 let plugin1 = actor {
12   insert(bst, key, value) {
13     bst<-insert(key, value)
14   }
15 }
16
17 let plugin2 = actor {
18   queryInsert(bst, key) {
19     bst<-query(key, object {
20       queried(value) {
21         if(value > 0) {
22           bst<-insert(key, value - 1)
23         }
24       }
25     });
26   }
27 }
```

Figure 2. A shared bst encapsulated in a separate actor

Figure 2 shows that the restriction of only being able to communicate asynchronously with a remote shared resource forces the programmer to structure his code in a very unintuitive way (CPS, lines 19–25). If we want to query and afterwards insert a new value in the BST to update it, we either have to extend the implementation of our BST with an `update` method or we combine the `query` and `insert` method in some way. Let us assume that changing the interface of the BST is not possible[3] and we need to employ the latter solution. Inter-actor communication always happens asynchronously in the event-loop model and therefore does not yield a return value. If we want to access items in our BST we will need a way to send back the result of the `query` method. The common approach to achieve this is to add an extra argument to each message that represents a callback. This client implements the continuation of our program given the return value of the message. In our example this is done via the `queried` method.

On line 19 we asynchronously send a `query` message to the BST passing a key as a parameter as well as a reference to an object that implements the continuation of our program given the return value of the `query` method (lines 20–24). Once the `bst` actor is processing the event it will eventually send back the result of the query to the client object via an asynchronous message (line 7). Because the client object was created by the `plugin 2` actor that message will then be scheduled as an event in the event-queue of the `plugin 2` actor. Note that while the `bst` actor is busy processing the query request, the `plugin 2` actor is available for handling other incoming events. Once the `plugin 2` actor is ready to process the "queried" event it can then decide whether or not to send an insert message to the bst actor depending on the return value of the query (lines 21–23).

The lack of synchronous communication with remote resources forces us to write our code in a CPS. Ideally we would want the

[3] This can be true for various reasons. Either legacy reasons or it might be that the query and insert messages need to be combined in a non-trivial way that also involves other remote objects.

`query` and `insert` method to be evaluated in the context of one event, which is not possible in either the event-loop actor model.

Extra synchronization conditions on groups of messages are not possible

In some cases it is possible that a certain interleaving of the evaluation of different messages leads to event-level race conditions. For example, in Figure 2 we introduce a race condition when both plugin 1 and plugin 2 try to insert a new value in the BST. Even if we would somehow avoid having to use CPS, any unwanted interleaving of the `query` an `insert` methods might lead plugin 2 to update the BST using old information. For example, if the bst actor first receives a query event from plugin 1, then the insert event from plugin 2 and only then the insert event from plugin 1, then plugin 1 updated the value of the bst depending on old information which is a race condition.

The reason race conditions like these occur when programming in an actor language is because different messages, sent by the same actor, cannot always be processed atomically. Programmers cannot specify extra synchronization conditions on groups of messages. A programmer is limited by the smallest unit of non-interleaved operations provided by the interface of the objects he or she is using and there are no mechanisms provided to eliminate unwanted interleaving without changing the implementation of the object (i.e. there are no means for client-side synchronization). There are ways to circumvent this, such as batch messages [25], but they do not solve the problem in the case where there are data dependencies between the different messages (e.g. in our example we need the value of the `query` method to be able to pass it to the `insert` method).

One way to solve this issue in our specific case would be to introduce a "coordination actor" that synchronizes access to the BST. Figure 3 illustrates how we could implement this.

The coordinator implements an asynchronous lock that can be acquired when the lock is available and released otherwise. Using a coordinator like this to guard critical sections has a number of disadvantages:

- Because all operations are asynchronous all of the actors will stay responsive to any message. However, this approach just reintroduces all the issues of traditional locking techniques. For example, similarly to deadlocks, progress can still be lost if different client objects are waiting to acquire a lock on a coordinator locked by the other client.
- Because the coordinator actor is a shared resource as well, asynchronous locking mechanism introduces another level of CPS code (lines 25 and 39).
- Introducing locks like this has the additional overhead of having to use the message passing system to both acquire and release a lock which makes it unsuitable for fine-grained locking.

No parallel reads

The main inefficiency of the actor model with respect to parallel programming is the fact that data cannot be read truly in parallel. This is assuming that we represent shared state as a separate actor. If we want to read (part of) an actor's state in parallel we have to go through the message passing system and the event-loop of the actor, which will handle each received event sequentially.

In Figure 2, our shared `bst` resource needs to be encapsulated by an actor. This means that all `query` messages will be needlessly sequentialized (in the absence of a `queryInsert` method)

Not only does this make accessing a large data structure from within different components of an application inefficient, it also makes it difficult to implement typical data-parallel algorithms efficiently within the actor model.

```
1  let bst = actor {
2    insert(key, value, client) {
3      result := ...
4      client<-inserted(result);
5    }
6    query(key, client) {
7      result := ...
8      client<-queried(result);
9    }
10 }
11
12 let coordinator = actor {
13   acquire(client) {
14     ...
15     client<-aqcuired();
16     ...
17   }
18   release() {
19     ...
20   }
21 }
22
23 let plugin1 = actor {
24   insert(coordinator, bst, key, value) {
25     coordinator<-acquire(object {
26       aqcuired() {
27         bst<-insert(key, value, object {
28           inserted(ignore) {
29             coordinator<-release();
30           }
31         });
32       }
33     });
34   }
35 }
36
37 let plugin2 = actor {
38   queryInsert(coordinator, bst, key) {
39     coordinator<-acquire(object {
40       aqcuired() {
41         bst<-query(key, object {
42           queried(value) {
43             if(value > 0) {
44               bst<-insert(key, value - 1, object {
45                 inserted(ignore) {
46                   coordinator<-release();
47                 }
48               });
49             } else {
50               coordinator<-release();
51             }
52           }
53         });
54       }
55     });
56   }
57 }
```

Figure 3. Synchronizing access to the BST

2.5 Our approach

Ideally we would want a third option in which we represent shared state as objects that do not belong to any particular actor but rather to a separate entity on which multiple actors can have synchronous access in a controlled way. This way we avoid all the issues with replicating state and also avoid all the issues that come with asynchronously communicating with that shared state.

3. The solution: Domains and views

Our approach allows the programmer to bundle any number of objects in the shared state as a domain. A domain does not belong to a specific actor but is rather a separate entity on which actors can have synchronous access. This synchronous access is important as most of the problems we identified are caused by the use of asynchronous communication to access the shared state. In our approach this synchronous access is represented by a "view". Views are a synchronization mechanism that allows one or more actors to have synchronous access to a shared domain for the duration of one event-loop event. There are two kinds of views, a shared and an exclusive view which mimic multiple reader, single writer access as a synchronization strategy.

As we discussed in section 2.4 an actor is a combination of an object heap and an event loop. The `actor{<expression>}` syntax creates a new event-loop and an object heap initialized with a single object initialized with `<expression>`. Evaluating the actor expression will result in a remote reference to that object. Similarly, a domain is just a container for a number of objects. An actor can never have a direct reference to a domain as a whole. Rather it can have references to objects inside that domain. From now on we will refer to these kinds of references as *domain references*. The `domain{<expression>}` syntax will create a new domain and initialize that domain's object heap with one object initialized by `<expression>`. Evaluating the domain expression will result in a domain reference to that object.

SHACL has a number of primitives to **asynchronously** request access rights to a particular domain using a domain reference. Once the corresponding domain becomes available for shared or exclusive access, an event is queued in the event-loop of the requesting actor. During that event, that actor has a window to synchronously access any object encapsulated by that domain using a domain reference.

Figure 4 gives a quick illustration of the usage of domains and views. Note that the `bst` actor of figure 2 has been replaced by a `domain`. As we saw in the previous section, the `domain` syntax on line 1 will create a new domain with a single object that implements two methods, `insert` and `query`. The return value of the domain syntax is always a domain reference to that object. This means that in our example the `bst` variable will contain a domain reference. Any object created by an expression nested inside the `domain` syntax cannot have access to variables that outside of the scope of that `domain`. The domain reference contained in the variable can be arbitrarily passed around between actors but can only be dereferenced when obtaining a view.

In Figure 4 sending a `queryInsert` message to plugin 2 will first asynchronously request an exclusive view on the bst domain reference (line 21). Once the corresponding domain becomes available for exclusive access, an event is scheduled in the event queue of the plugin 2 actor which will evaluate the block of code provided to the `whenExclusive` primitive (lines 22–25). Note that this block of code is executed by the actor that created it (`plugin2`) and it has access to all lexically available variables such as `bst` and `key`. The plugin 2 actor can synchronously access the BST within that block of code. It can synchronously query it for a certain key and then synchronously update that key-value pair depending on the result of that operation. The same holds if we want to read the same value multiple times, read and/or update different values, etc. Additionally, during the event on which we acquired the view the actor code no longer has to be written in CPS to read and/or write values from and to our shared resource, we can synchronize different messages to the same resource and in the case of a shared view we can even parallelize reads to that resource.

```
1  let bst = domain {
2    insert(key, value) {
3      ...
4    }
5    query(key) {
6      ...
7    }
8  }
9
10 let plugin1 = actor {
11   insert(bst, key, value) {
12     whenExclusive(bst) {
13       bst.insert(key, value);
14     }
15   }
16 }
17
18
19 let plugin2 = actor {
20   queryInsert(bst, key) {
21     whenExclusive(bst) {
22       value := bst.query(key);
23       if(value > 0) {
24         bst.insert(key, value - 1)
25       }
26     }
27   }
28 }
```

Figure 4. Illustration of domains and views

3.1 View primitives

In this section we will only consider view primitives that acquire a view on a single domain at a time. For SHACL this set of primitives was extended to also allow acquiring shared and/or exclusive views on a set of domains (See section 4). SHACL supports the following primitives for requesting views on a domain reference:

$$\text{whenShared}(e)\{e'\}$$
$$\text{whenExclusive}(e)\{e'\}$$

Here, e is a valid SHACL expression that evaluates to a domain reference and e' is any valid SHACL expression. Note that these primitives are asynchronous operations, they will schedule a view-request and immediately return. After the request is scheduled, the event-loop of the actor can resume processing other events in its event-queue. Once the domain becomes available two things happen. First the domain is locked for exclusive or shared access. Then an event that is responsible for evaluating the expression e' is put in the event-queue of the corresponding actor. Once that event is processed the domain is freed again, allowing other actors to access it.

Figure 5 illustrates how views are created. Both actor `A` and actor `B` have a reference to the shared object. If they want to access this shared object, first they need to request a view on that object. Attempting to access a domain reference outside of a view results in an error. Once the request is handled by the domain, any reference to an object inside that domain becomes synchronously available for the duration of one event. When e' is evaluated, the actor loses its access rights to that domain. A shared view allows the actor to synchronously invoke read-only methods of all the objects within the corresponding domain. Any attempt to write a field of a domain object during a shared view will result in an error. An exclusive view allows the actor to synchronously invoke any

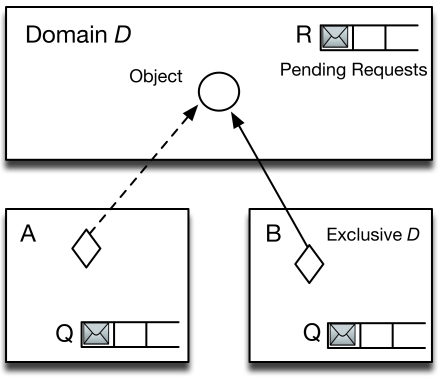

Figure 5. Actor A and B share a reference to an object inside domain D on which Actor B has an exclusive view.

method on objects inside the corresponding domain, regardless of whether they change the state of the object(s) inside that domain.

3.2 Semantic properties

In this section we will evaluate and discuss our approach with regard to the original actor model. The following topics will be discussed: deadlock freedom, race condition freedom and macro-step semantics.

3.2.1 Deadlock freedom

The absence of deadlocks and low-level race conditions are the two properties of the actor model that differentiate it from lower-level models and provide the required guarantees to build large concurrent applications in a sustainable manner. To maintain deadlock-freedom, the following two restrictions are enforced for views:

View requests are non-blocking. All primitives that request views are non-blocking. As explained in Section 3.1, the request for a view is scheduled as an asynchronous event which is processed only once the domain becomes available. This implies that all operations in our language terminate, meaning that all events can be processed in a finite operational time (if programmed correctly). And thus, the event for which a lock was acquired will eventually terminate and release the lock again, which is important to ensure that our language remains deadlock free.

A view on a domain only exists for the duration of one event. Events in our model can be considered atomic operations with a finite operational time. This means that any domain that is currently unavailable due to a view will become available at some point in the future.

With those restrictions in place SHACL is guaranteed to be deadlock free. With view requests being non-blocking, and the absence of any other blocking operation in the model, it is guaranteed that *wait-for* cycles can not be constructed with the basic primitives provided.

As discussed in Section 3.1, views are only held for the duration of a single event, and requesting a "nested" view while holding another view is an asynchronous operation. All this supports the notion of an event being executed as an atomic operation with a finite number of operational steps. Barring any sequential infinite loops included by the programmer.

3.2.2 Race condition freedom

To maintain race condition freedom, the following three restrictions are enforced for views:

Only allow view requests on domain references All our primitives require that the reference on which a view is requested is a domain reference. In contrast with other remote objects, domain objects are not allowed to have direct references to objects contained in another domain. This way, accidentally shared state is avoided, which ensures that no low-level race conditions can occur in our model. Without this restriction, multiple actors could access the same free variable in the lexical scope of a domain object, opening the door for classical data races.

Domain references cannot be accessed outside of a view Care has to be taken of views which had been previously lexically captured but haven been only available in an asynchronous operation. Since the lexical scope would hold a reference to a domain object that is not protected by a view anymore, race conditions could be introduced. To avoid such data races, any attempt to access a domain object outside of a view throws an error.

Object creation inside a domain. Any object creation expression lexically nested inside a domain generates objects owned by that domain. Views are acquired on a domain and dereferencing an object owned by that domain can never expose that domain's content. As such, any reference to an object that is owned by a domain is always a domain reference.

These three rules ensures that, in any scenario, any domain reference or lexically nested state is no longer accessible while processing later events without requesting a new view. They also ensure that any concurrent updates of shared state are impossible and thus ensures that we avoid race conditions by construction.

3.2.3 Macro-step semantics

The actor model provides one important property for formal reasoning about different program properties. This property is the macro-step semantics [4].

In an actor model, the granularity of reasoning is a message/event. For the properties of a program, each event is processed in a single atomic step. This leads to a convenient reduction of the overall state-space that has to be regarded in the process of formal reasoning. Furthermore, this property is directly beneficial to application programmers as well in their development process. Programmers can design the semantics of message sends as coarse-grained as appropriate, reducing the potential problematic interactions.

After introducing domains and views, the question is whether the macro-step semantics still holds. Arguable, this is still the case, since the macro-step semantics only requires the atomicity of the evaluation of a message, but does not imply any locality of changes. Thus, changing the state of an object for which a view was obtained does not violate atomicity, since we only allow exclusive views for state modifications. Shared views for reading state are also not violating the semantics since state is not actually changed.

Based on this reasoning, an actor-model with the concept of views presented in Section 3.1 still maintains the macro-step semantics, and thus keeps the main properties of the actor model that are beneficial for formal reasoning intact.

Furthermore, the semantics of all writes in our model, being restricted either to local writes inside an actor, or writes protected by an exclusive view, result in a memory model which enforces sequential consistency [11]. Thus, the original semantics remains preserved and allows the application of relevant reasoning techniques.

3.3 Expressiveness

Since the actor model relies solely on asynchronous event processing to avoid deadlocks, the expressiveness of such a language is typically impaired.

With the mechanisms proposed here, it is however possible to grant synchronous access to domain objects protected by views. Listing 4 introduced the corresponding example and demonstrates how to access a shared resource synchronously, which could not be expressed before. For this specific case, the alternative solution would be to change the interface of the remote object, to be able to request and update its state in a single step. However, that approach is neither always possible, e. g., for third-party code, nor desirable. Also, this solution would not be appropriate for synchronizing updates to different domain objects as ensuring synchronized access in the traditional actor model would require to bundle these objects into one actor. Thus, with views the expressiveness is extended considerably over the standard actor model. Programmers can model their shared state without taking into account how this shared state will be accessed from the client side and the client side can synchronize and compose access to different objects in an arbitrary way.

3.4 Conclusion

In this section we have shown that with the use of views we can avoid the problems discussed in section 2. Firstly, by **not replicating** the shared state we avoid the need to keep replicas consistent. Secondly, from within a view we do **not** need to employ **CPS** to access shared state. If we have synchronous access to the domain object we can directly access it's fields without using the message passing system. Thirdly, we can safely build more coarse-grained synchronization boundaries by combining messages to objects within the same domain in an arbitrary way during the event in which we acquired the view. Lastly, if we only use shared views on a resource we can **read** from that resource **in parallel**.

4. SHACL further features

Section 3 discussed only the core features of SHACL. In this section we will discuss a number of other important features of SHACL.

4.1 Views on multiple domains

Currently, SHACL only supports shared and exclusive views, which mimic single writer, multiple reader locking. This means that it is impossible to do parallel updates of objects a single domain. A workaround for this problem would be to subdivide the shared data structure into several domains. We could for example put each node of the binary search tree of our example in a separate domain. This however also means that any parallel updates to that data structure need to be synchronized by the program. SHACL has a primitive that allows the programmer to synchronize access to multiple domains:

$$\text{when}(e, e'')\{e''\}$$

The `when` primitive takes any 2 SHACL expressions e and e' that evaluate to two arrays of domain references. The first array has to contain all the domain references for which the programmer wants to have shared access and similarly the second array has to contain all the domain references for which the programmer wants to have exclusive access. e'' is the expression that will be scheduled as an event in the event-loop of the executing actor once all the necessary resources become available.

In SHACL there is a global ordering in which all domains are locked for shared and/or exclusive access. This is to prevent deadlocks when views are requested on multiple domains.

4.2 Futures

In section 1 we already mentioned that SHACL supports future-type messages. Futures introduce a synchronization mechanism for actors to synchronize on the reception of a message without using callbacks. Traditional asynchronous messages have no return value. A developer needs to work around this lack of return values by means of an explicit customer object as seen in all the examples throughout the paper. Future type messages allow the programmer to hide this explicit callback parameter.

In contrast to regular asynchronous messages, a future-type message does have a return value. It returns a future-value that represents the "eventual" return value of the message that was sent. The developer can then register an observer with that future-value using a special `whenBecomes` primitive. When the original message is processed by the receiving actor, the future is "resolved" with the return value of that message and any registered observer is notified. A notified observer triggers an event that is scheduled in the event-loop of the actor that executed the `whenBecomes` primitive.

The following example illustrates the usage of futures:

```
1  let cell = object {
2    c := 0;
3    get() {
4      c;
5    }
6    set(n) {
7      this.c := n;
8    }
9  }
10
11 let a = actor {
12   increase(counter) {
13         future := counter<-get();
14     whenBecomes(future -> c) {
15       counter<-set(c + 1);
16     }
17   }
18 }
19
20 a<-increase(cell);
```

Figure 6. Illustration of futures

In Figure 6 `get` is sent as a future-type message to the remote reference counter and immediately returns a future-value. An event is registered with that future that is responsible for updating the counter by sending it a regular asynchronous `set` message. Notice that using futures does not solve the issues discussed in section 2. We still need to employ CPS if we want to access several values of our remote object, event-level data races can still occur and reads are not parallelized.

The reason that futures are interesting for our model is because they work well together with domains and views. In fact, a view request in SHACL returns a future-value on which can be synchronized. If part of our computation depends on the atomic update of a shared resource but does not necessarily require synchronous access to that resource, these futures can be used to schedule code that can be executed after the view was released.

There is also a mechanism in SHACL to group futures into a single future (namely the primitive `group`). This mechanism can be used in conjunction with future-type messages to branch work to other actors and then synchronize on all of them. Or it can be

used in conjunction with domains and views to schedule a number of atomic updates and then synchronize on the completion of all of them.

5. Related work

The engineering benefits of semantically coarse-grained synchronization mechanisms in general [10] and the restrictions of the actor model [16] have been recognized by others. In particular the notion of domains and *view*-like constructs has been proposed before.

Demsky and Lam [10] propose views as a coarse-grained locking mechanism for concurrent Java objects. Their approach is based on static view definitions from which at compile time the correct locking strategy is derived. Furthermore, their compiler detects a number of problems during compilation which can aid the developer to refine the static view definitions. For instance they detect when a developer violates the view semantics by acquiring a read view but writing to a field. The main distinction between our and their approach comes from the different underlying concurrency models. Since Demsky and Lam start from a shared-memory model, they have to tackle many problems that do not exist in the actor model. This results in a more complex solution with weaker overall guarantees than what our approach provides. First of all, accessing shared state without the use of views is not prohibited by the compiler thereby compromising any general assumptions about thread safety. Secondly, the programmer is required to manually list all the incompatibilities between the different views. While the compiler does check for inconsistencies when acquiring views, it does not automatically check if different views are incompatible. Forgetting to list an incompatibility between different views again compromises thread safety. Thirdly, acquiring a view is a blocking statement and nested views are allowed, possibly leading to deadlocks. They do recognize this problem and partially solve this by allowing simultaneously acquiring different views to avoid this issue. But avoiding the acquiring of nested views is not enforced by the compiler. Finally, their approach does not support a dynamic notion of a view which could be used to safely access shared state depending on runtime information.

Hoffman et al. [14] show the need for programs to isolate state between different subcomponents of an application. They propose protection domains and ribbons as an extension to Java. Similarly to our approach, protection domains dynamically limit access to shared state from different executing threads. Access rights are defined with ribbons where different threads are grouped into. While their approach is very similar to ours, they started from a model with less restrictions (threads) and built on top of that while we started from the actor model which already has the necessary isolation of processes by default. While access modifiers on protection domains do limit the number of critical operations in which race conditions need to be considered. If two threads have write access to the same data structure, access to that data structure still needs to be synchronized.

Axum [17] is an actor based language that also introduced the concept of domains for state sharing. Similarly to our approach single writer, multiple reader access is provided to domains. Access patterns in Axum have to be statically written down, which does give some static guarantees about the program but ultimately suffers from the same problems as the views abstractions from Demsky and Lam. Although the Axum project was concluded it also showed that there is an interest in a high level concurrency model that allows structuring interactive and independent components of an application.

ProActive [7] is middleware for Java that provides an actor abstraction on top of threads. It provides the notion of *Coordination actors* to avoid race conditions similar to views. However, the overall reasoning about thread safety is hampered since its use is not enforced. Furthermore, coordination actors are proxy objects that sequentialize access to a shared resource, and thus, are not able to support parallel reads, one of the main issues tackled with our approach. In addition, it is neither possible to add synchronization constraints on batches of messages, nor is deadlock-freedom guaranteed, since accessing a shared resource through a proxy is a blocking operation.

In Deterministic Parallel Java [8] the programmer has to use effect annotations to determine what parts (*regions*) of the heap a certain method accesses. They ensure race condition free programs by only allowing nested calls to write disjoint sub-regions of that region. This means that this approach is best suited for algorithms that employ a divide and conquer strategy. In our approach we want a solution that is applicable to a wider range of problems including algorithms that randomly access data from different regions.

Parallel Actor Monitors [19] (PAM) is a related approach to enable parallelism inside a single actor by evaluating different messages in the message queue of an actor in parallel. The difference with our approach is that the actor that owns the shared datastructure is still the only one that has synchronous access on that resource. In our approach we apply an inversion of control where the user of the shared resource has exclusive access instead of the owner. This inversion of control allows an actor in SHACL to synchronize access to multiple resources which is not possible in the case of PAM.

6. Conclusion

The Actor Model is a good model for concurrent programming, it provides a number of safety guarantees for issues that are often problematic in other models (Deadlock freedom, data-race freedom, macro-step semantics). Unfortunately the restrictions on this model often limit the expressiveness of the model in comparison with less strict implementations, limiting its adoptability as a mainstream programming model. The issue of accessing shared state is one shared between all actor languages. Others solve this issue by allowing the programmer to break actor boundaries as an escape hatch (e.g. Scala). In this case, the programmer has to rely on traditional locking mechanisms to synchronize access to that state, reintroducing all problems that come with locks. Others combine several concurrency models to solve this issue. For example, Clojure [13] both implements actor based concurrency primitives as well as a Software Transactional Memory. In our approach we tried to tailor our solution specifically for the Actor Model ensuring maximum interoperability between the different primitives. The advantages of our model over the traditional event-loop model are threefold. Firstly we avoid the continuation passing style of programming when accessing shared state. Secondlyndly we allow the programmer to introduce extra synchronization constraints on groups of messages and lastly we are able to model true parallel reads.

7. Acknowledgements

Joeri De Koster is supported by a doctoral scholarship granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO)

References

[1] Akka. `http://akka.io/`.

[2] Asyncobjects framework. `http://asyncobjects.sourceforge.net/`.

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. *AITR-844*, 1985.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[5] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. Concurrent programming in erlang. 1996.

[6] M. Astley. The actor foundry: A java-based actor programming environment. *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.

[7] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[8] R. Bocchino Jr, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. *ACM SIGPLAN Notices*, 44(10):97–116, 2009.

[9] E. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 19, pages 7–10, 2000.

[10] B. Demsky and P. Lam. Views: Object-inspired concurrency control. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 395–404. ACM, 2010.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News*, 18:15–26, May 1990. ISSN 0163-5964.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3): 202–220, 2009.

[13] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[14] K. Hoffman, H. Metzger, and P. Eugster. Ribbons: a partially shared memory programming model. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 289–306. ACM, 2011.

[15] G. Hohpe. Programming Without a Call Stack–Event-driven Architectures. *Objekt Spektrum*, 2006.

[16] R. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[17] Microsoft Corporation. Axum programming language. `http://tinyurl.com/r5e558`.

[18] M. S. Miller, E. D. Tribble, J. Shapiro, and H. P. Laboratories. Concurrency among strangers: Programming in e as plan coordination. In *In Trustworthy Global Computing, International Symposium, TGC 2005*, pages 195–229. Springer, 2005.

[19] C. Scholliers, É. Tanter, and W. De Meuter. Parallel actor monitors. Technical report, 2010. vub-tr-soft-10-05.

[20] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. *ECOOP 2008–Object-Oriented Programming*, pages 104–128, 2008.

[21] H. Sutter. Welcome to the jungle. http://herbsutter.com/welcome-to-the-jungle/, 2011.

[22] D. Ungar and R. Smith. *Self: The power of simplicity*, volume 22. ACM, 1987.

[23] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*, pages 3–12. Ieee, 2007.

[24] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[25] A. Yonezawa, J. Briot, and E. Shibayama. Object-oriented concurrent programming ABCL/1. *ACM SIGPLAN Notices*, 21(11):268, 1986.