Domains: safe sharing among actors☆

Joeri De Koster, Stefan Marr, Theo D'Hondt, Tom Van Cutsem

Vrije Universiteit Brussel,
Pleinlaan 2,
B-1050 Brussels, Belgium

Abstract

The actor model is a concurrency model that avoids issues such as deadlocks and data races by construction, and thus facilitates concurrent programming. While it has mainly been used for expressing distributed computations, it is equally useful for modeling concurrent computations in a single shared memory machine. In component based software, the actor model lends itself to divide the components naturally over different actors and use message-passing concurrency for the interaction between these components. The tradeoff is that the actor model sacrifices expressiveness and efficiency with respect to parallel access to shared state.

This paper gives an overview of the disadvantages of the actor model when trying to express 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 data races.

Keywords: Actor Model, Domains, Synchronization, Shared State, Race-free Mutation

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 [1]. 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.

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 [2], E [3], AmbientTalk [4], SALSA [5] and Kilim [6]. The major benefit of pure actor languages is that the developer gets strong safety guarantees: low-level data races are ruled out by design. The drawback is that such pure

☆This paper is an extension of: De Koster, J.; Van Cutsem, T. & D'Hondt, T. (2012), Domains: safe sharing among actors, in 'Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions', pp. 11–22.

Email addresses: jdekoste@vub.ac.be (Joeri De Koster), smarr@vub.ac.be (Stefan Marr), tjdhondt@vub.ac.be (Theo D'Hondt), tvcutsem@vub.ac.be (Tom Van Cutsem)

actor 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 3.3. Actor libraries typically do not share these restrictions but also do not provide the strong safety guarantees because these libraries are often for languages whose concurrency models are based on shared-memory multithreading. Examples for Java include ActorFoundry [7] and AsyncObjects [8].

Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [9] and Akka [10]. What these libraries have in common is that they do not enforce actor isolation, i. e., they do not guarantee that actors do not share mutable state. On the other hand, it is 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, 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:

Safety The isolation between actors structures programs and thereby facilitates reasoning 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 safety and liveness invariants such as deadlock and race-condition freedom of the "core" application. Thus, as in pure actor languages, we want an actor system that maintains strong language-enforced guarantees and prevents low-level data races and deadlocks by design.

Expressiveness Many phenomena in the real world can be naturally modeled using message-passing concurrency, for instance telephone calls, e-mail, digital circuits, and discrete-event simulations. Sometimes, however, a phenomenon can be modeled 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 read/write access to shared mutable state, without having to encode shared state as encapsulated state of a shared actor. Furthermore, by enabling direct *synchronous* access to shared state, i. e., without requiring a message send, we gain stronger synchronization constraints and prevent the inversion of control that is characteristic for interacting with actors, as interaction is asynchronous.

Efficiency Today, multicore hardware is becoming the prevalent computing platform, both on the client and the server [11]. While multiple isolated actors can be executed perfectly in parallel by different hardware threads, shared access to a single actor can still form a serious sequential bottleneck. In pure actor languages, shared mutable state is modeled with a specific actor and all requests sent to it are 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 Shacl language is available online[2]. This paper also provides an operational semantics for a small subset of our language named Shacl-Lite. This operational semantics serves as a complete and detailed specification of our model.

The rest of this paper is structured as follows, Section 2 gives a short overview of the communicating event-loops model on which the Shacl model was based. In Section 3, we present a number of problems that occur when representing shared state in that model. Section 4 presents domain and view abstractions as an extension to actors. In Section 5, a formal specification of our model is given by means of an operational

[1]Pronounce as "shackle", short for **sh**ared **ac**tor **l**anguage
[2]`http://soft.vub.ac.be/~jdekoste/shacl`

semantics. Section 6 lists a number of important additional features of Shacl. The related work section and our conclusion complete the paper.

2. Communicating event-loops

Before we explain the issues with state sharing in message-passing concurrency models based on actors, we first provide more details about the precise nature of the actor model upon which Shacl is based.

The concurrency model of Shacl is based on the communicating event-loops model of E [3] and AmbientTalk [4] where actors are represented by *vats*. Each vat/actor has a single thread of execution, an object heap and an event queue. Generally, when actors are first introduced to one another, they need to exchange addresses. In the event-loop actor model such an address is always in the form of a remote reference to an object. The referenced object then defines how another actor can interface with that actor. The big difference between communicating event-loops and traditional actor languages is that traditional actor languages usually only provide a single entry point or address to an actor. An event-loop actor can define multiple objects and hand out different references to those objects.

Each object in an actor's object heap is *owned* by that actor. Within an actor, references to objects owned by that same actor are called *local references*. References to objects owned by other actors are called *remote references*. Actors are not first class entities and do not send messages to each other directly. Instead, objects *owned* by different actors send asynchronous messages to each other using remote references to objects inside another actor. An asynchronous message sent to an object in another actor is enqueued as an incoming event in the event queue of the actor that owns the receiver object and immediately returns without a result (see Section 6.1.3). The thread of execution of that actor is an event-loop that perpetually takes events from its event queue and delivers them to the local receiver object. Hence, events are processed one by one. The processing of a single event is called a *turn*, and each turn processes one event to completion.

Within a single actor, synchronous communication can be used to invoke any method that is pointed to by the local reference. A synchronously invoked method is executed within the same actor and returns its result immediately. Any attempt to synchronously access a remote reference is considered to be an erroneous operation.

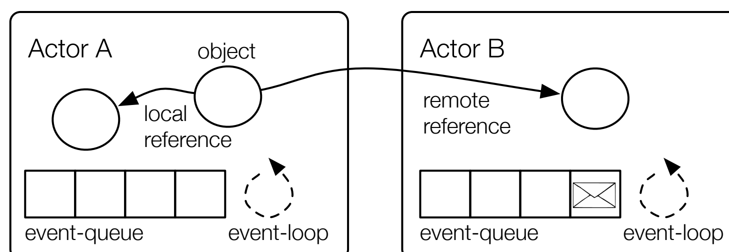

Figure 1: The event-loop actor model

Figure 1 is an illustration of this model. In this figure, actors are represented by boxes containing objects (represented by the circles), an event-queue, and an event-loop. The arrows represent references from one object to another. A reference from one object to another object in the same actor is called a local reference. A reference from one object to another object in a different actor is called a remote reference. Currently, actor B has a single event scheduled in its event-queue.

Shacl is a minimal implementation of a pure event-loop actor language. The sequential subset of Shacl is a simple dynamically typed object-based language (similar in style to Self [12] and JavaScript).

In the next section, we explain the issues with state sharing that occur given the restrictions of this pure communicating event-loops model. Later, we extend Shacl with a new abstraction, called a *domain*, that allows controlled synchronous access to shared state between actors (Section 4).

3. The problem: Accessing non-local shared state

Event loop actors are isolated, i. e., they have no direct access to each others' state. If shared state is required, it must be represented either by replicating the shared state over the different actors or by encapsulating the shared state in an additional independent actor. In this section we discuss the disadvantages of both approaches using a motivating example.

3.1. Motivating example

As a motivating example, consider an application with a plugin architecture. A plugin architecture enables third-party developers to extend the applications with plug-ins that provide additional functionality. Isolation and encapsulation are important properties for such architectures to guarantee the integrity of the different plugins. And more importantly, they guarantee the integrity of the core application itself. A crashing plug-in should not crash the main application. Hence, the event-loop actor model is a good fit for such application as it already enforces these properties at the language level. Actors can be used to model the different plugins and 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. Such a resource may be globally available for all plugins or just shared between a subset of the plugins.

For example, plugin-oriented browser applications might require shared access to the *document object model*[3] (DOM) of a webpage. A DOM is the internal representation of all the elements of a webpage and can be referenced and modified by the browser and its different plugins. In such a scenario, a layouting plugin can require write access to the DOM to modify the layout of a web page while the module that is responsible for visualizing the web page on the screen only requires read access to the DOM. Another example of such an application could be Google's or Microsoft's maps application, which could use incremental route planning algorithms that display their progress on the maps. Other examples are modern online editors for emails, text documents, or even code. They can have multiple plugins for instance for simple spellchecking, parsing and syntax highlighting, or type checking and other complex static analysis. Such plugins could work in the background but need to modify the DOM to display their results. In these cases it is important for these different plugins to synchronize the access to the DOM to make sure that updates such as removing or adding elements to a web page are done in a consistent manner.

These are only a few concrete examples, but there are many other cases where a modular synchronization mechanism is necessary. A shared tree data structure such as a DOM tree is just one application scenario. For this paper, we consider a simplified variant of this scenario and our examples use two plugins that require access to a shared binary search tree (BST) data structure.

Figure 2 shows how such an application could be modeled using an UML class diagram. There are two different plugins that use this tree 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 the `insert` method. 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.

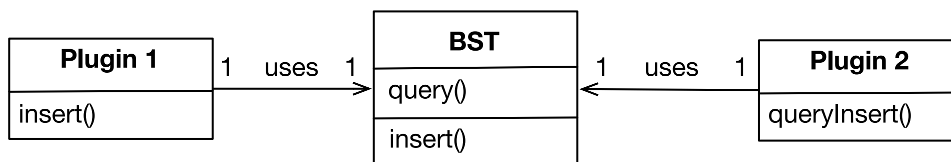

Figure 2: Two plugins using the same shared resource

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

[3]http://www.w3.org/TR/domcore/

3.2. Replication

One option for representing shared state in the actor model is to replicate this state inside different actors that require access to it. For our example this means that the BST needs to be replicated in both plugin 1 and 2. This approach has disadvantages with regard to consistency, memory overhead, and performance:

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 is no longer feasible. Lowering the availability or consistency of these replicas can be a solution to this problem [13]. Unfortunately, whether lowering either availability or consistency is possible is application dependent. In general, keeping replicated state consistent is a hard problem that usually introduces unnecessary code.

Memory usage Replication increases the memory usage 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.

Copying cost In certain applications, 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 example, calculating the sum of all the elements in a vector in parallel would be less efficient than the sequential version if we first need to copy each subset of the vector to a different actor.

3.3. 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: Firstly, the asynchronous communication between the actors encapsulating the state forces **continuation-passing style** upon the programmer. Secondly, it is impossible to put **synchronization conditions** on groups of messages that are sent to the encapsulating actor. And finally, the semantics of actors ensure that there are **no parallel reads**, consequently sequentializing all parallel accesses from different actors independent of whether these accesses would be conceptually safe or not. This section explains these issues using the following example:

Figure 3 shows an implementation of the BST that is encapsulated by a separate actor. Similarly to E [3] and AmbientTalk [4] in SHACL the actor$\{\overline{f:e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}$ syntax evaluates to a new actor with its own separate object heap and event loop. That object heap is then initialised with a single new object for which the fields $(\overline{f:e})$ and methods $(\overline{\mathrm{m}(\overline{x})\{e\}})$ are defined by the body of the `actor` syntax. The `actor` expression immediately evaluates to a remote reference to that newly created object. Similarly, the object$\{\overline{f:e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}$ syntax evaluates to a newly created object initialised with a number of fields and methods. The `object` expression immediately evaluates to a local reference to the newly created object. SHACL uses the dot (`.`) and arrow (`<-`) notation to syntactically distinguish between synchronous and asynchronous communication. 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. 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 process 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.

Asynchronous communication leads to continuation-passing style

The style of programming where the control of the program is passed around explicitly as a continuation is called continuation-passing style (CPS), also known as programming without a call stack [14]. The

```
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 let plugin2 = actor {
17   queryInsert(bst, key) {
18     bst<-query(key, object {
19       queried(value) {
20         if(value > 0) {
21           bst<-insert(key, value - 1)
22         }
23       }
24     });
25   }
26 }
```

Figure 3: A shared binary tree data structure encapsulated in a separate actor

problem of having to use CPS to access a remote resource is common and can be found in various other actor languages like SALSA [5], Kilim [6], etc. The problem with this style of programming is that it leads to "inversion of control" [15].

Figure 3 shows that the restriction of only being able to communicate asynchronously with a remote shared resource forces the programmer to structure the code using CPS (lines 18–25). The example creates three actors called `bst`, `plugin1`, and `plugin2`. If we want to query and afterwards insert a new value in the binary search tree represented by the `bst` actor, we either have to extend the implementation of it with an `update` method or we combine the `query` and `insert` method in some way. Let us assume that changing the interface of `bst` is not possible[4] 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 of the binary search tree 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 by passing an object as a callback, where the object has a single `queried` method to process the result.

On line 18 we asynchronously send a `query` message to `bst` passing a key as a parameter as well as a reference to an object that represents 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 `plugin2` actor that message will then be scheduled as an event in the event queue of the `plugin2` actor. Note that while the `bst` actor is busy processing the query request, the `plugin2` actor is available for handling other incoming events. Once the `plugin2` 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 continuation-passing style. Ideally, we would want the `query` and `insert` method to be called synchronously on `bst` in the context of one event, which is not possible in 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 bad message interleavings. For example, in Figure 3 we introduce a race condition when both plugin 1 and

[4] This can be true for various reasons. The `bst` actor may be defined as part of library code 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.

plugin 2 try to insert a new value into the binary tree. Even if we would somehow avoid having to use CPS, any unwanted interleaving of the `query` and `insert` methods might lead plugin 2 to update the binary tree 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 binary tree based on old information, which is a bad message interleaving.

Bad interleavings like these occur when programming in an actor language 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 [16], but they do not solve the problem in the case when there are data dependencies between the different messages. For instance in our example we need the result 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 controls access to the `bst` actor. Figure 4 illustrates, using an UML communication diagram, how we could implement this. In this figure we make an extension of the code in Figure 3 where we add a coordinator actor that is responsible for controlling access to the `bst` actor.

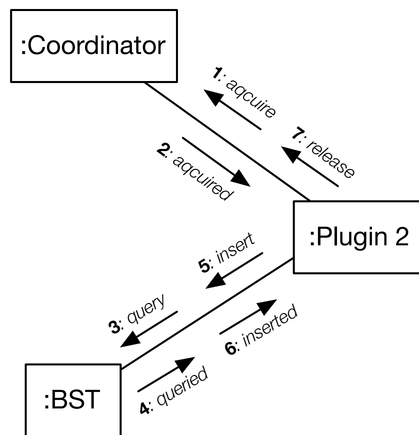

Figure 4: Controlling access to the binary search tree (BST) actor using a coordinator actor.

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 reintroduces all the issues of traditional locking techniques. For example, similarly to deadlocks, execution can stall 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. All messages between the different actors have to be sequentialized in the order depicted in Figure 4. This means that each of those messages introduces another level of CPS.

- Introducing locks like this has the additional overhead of having to use the message-passing system to both acquire and release a lock. This overhead makes this locking technique 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 one actor wants to read (part of) another actor's state in parallel it has to schedule a message in the message queue of the actor, which will handle each received message sequentially.

In Figure 3, the shared binary search tree needs to be encapsulated by the `bst` actor. This means that all `query` messages will be needlessly sequentialized, because the interface does not include for instance 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 (e. g. parallel for, map-reduce).

4. The solution: Domains and views

We present 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, a domain, on which multiple actors can have synchronous access in a controlled way. This approach avoids the issues discussed in Section 3, which are caused by relying on state replication and having exclusively asynchronous access to shared state.

Our approach allows programmers to bundle any number of objects that are to be shared between actors as a *domain*. A domain is a container for objects that does not belong to a specific actor. Rather, it is a separate entity on which actors can acquire a *view*. A view grants an actor either *exclusive* or *shared* access to a domain and thus, synchronizes the domain access with other actors. The view allows the actor that acquired it to have synchronous access to all the objects inside the domain for the duration of a single turn, i. e., while processing one incoming message, which maintains the expected guarantees of an actor language. The two kinds of views, i. e., the *shared* and *exclusive* views mimic multiple-reader/single-writer access as a synchronization strategy. Thus, a *shared* view enables synchronous read access on the domain, while an *exclusive* view also provides synchronous write access. This synchronous access is important as most of the problems we identified are caused by the use of asynchronous message sends to access the shared state.

As we discussed in Section 3.3 an actor is a combination of an object heap and an event loop. The $\text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}$ syntax (cf. Figure 3) creates a new event loop and an object heap initialised with a single object initialised by the body of the `actor` syntax. Evaluating this actor expression results 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, in other words, domains are not first class entities. Rather an actor can have references to objects inside that domain. From now on, we will refer to these kinds of references as *domain references*. The $\text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}$ syntax will create a new domain and initialise that domain's object heap with one object initialised by the body of the `domain` syntax. Evaluating the domain expression results in a domain reference to that object.

Shacl has a number of primitives to *asynchronously* request access rights to a particular domain using the domain reference of an object contained within it. Once a request was granted and the corresponding domain becomes available for shared or exclusive access, an event is queued in the event queue of the requesting actor. During the turn processing this event, the actor has a temporary view to synchronously access any object encapsulated by that domain using the objects' domain references.

Figure 5 illustrates the usage of domains and views. Note that the `bst` actor in Figure 3 has been replaced by a `domain`. The `domain` expression 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` expression 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` expression cannot refer to variables outside of the lexical scope of that `domain`. In other words, the `domain` expression is a boundary for what variables are lexically available. A domain reference can be arbitrarily passed around between actors but can only be dereferenced while having acquired a view on its associated domain. Any `object` expressions that are evaluated in the context of a domain also belong to that domain. This means that if the `insert` method of our example contains code to create new objects, those newly created objects will always belong

to the encapsulating domain, regardless of what actor is executing that code. Similarly to actors, object ownership is lexically determined (see Section 5 for a more detailed explanation).

```
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 let plugin2 = actor {
18   queryInsert(bst, key) {
19     whenExclusive(bst) {
20       let value = bst.query(key);
21       if(value > 0) {
22         bst.insert(key, value - 1)
23       }
24     }
25   }
26 }
```

Figure 5: Illustration of domains and views

In Figure 5 sending a `queryInsert` message to `plugin2` will first asynchronously request an exclusive view on the `bst` domain (line 19). Once the corresponding domain becomes available for exclusive access, an event is scheduled in the event queue of the `plugin2` actor, which will evaluate the block of code provided to the `whenExclusive` primitive (lines 20–23) as a normal turn. 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 `plugin2` actor can synchronously access the binary search tree referenced by `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 turn in 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 the shared resource. Because the `whenExclusive` primitive is an asynchronous operation the programmer still needs to introduce a single level of CPS. However, once the view has been acquired, and the domain can be accessed synchronously, the use of CPS to structure the code is no longer needed. In Figure 3 two levels of CPS were needed to `query` and afterwards `insert` into the BST while in Figure 5 only a single level of CPS was needed to acquire the view. Without views the number of times CPS needs to be applied typically grows with the number of operations that need to be performed on the shared object. With views, this can be limited to only a single level of CPS. In addition to avoiding CPS, we can synchronize different messages directed at the same resource. Because we are guaranteed exclusive access for the duration of the view we know that the call to the `query` and `insert` methods inside the exclusive view will not be interleaved with other operations on the BST. Finally, while shared views were not used in this example, in the case of a shared view we can even parallelize reads of the shared resource.

4.1. View primitives

In this section we only consider view primitives that acquire a view on a single domain at a time. Shacl provides additional primitives to acquire shared and/or exclusive views on a set of domains (see Section 6). In order to keep semantics in Section 5 minimal, we treat these additional primitives as extensions. However the extension still preserves the guarantees given by Shacl.

Shacl supports the following primitives for requesting views on a single 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. A view-request, either shared or exclusive, is a request of an actor to acquire access to a particular domain. This view-request is scheduled in a view-scheduler by the actor executing the request. The view-scheduler is a passive entity. When a view is requested or released, the actor requesting or releasing the view will put itself in the queue or schedule the execution of the next view. 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 actor that requested the view. Once that event is processed (i. e. the expression e' is fully evaluated) the domain is freed again, allowing other actors to access it.

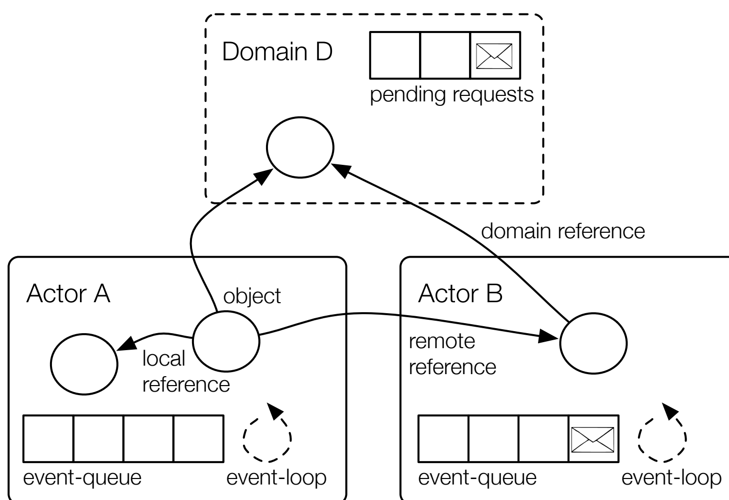

Figure 6: Different objects inside actors A and B share a reference to an object inside domain D.

Figure 6 is an extended version of Figure 1 and illustrates how domains are modeled. In this figure a domain, `D`, represented by the dotted box was added. This domain contains a single object, represented by a circle, and a set of pending requests. Objects in both actor `A` and actor `B` have a reference to the shared object. A reference from one object to another object inside a domain is called a domain reference. If the objects in actor A and B want to access this shared object, first they need to request a view on the domain of that object. Attempting to access a domain reference outside of a view results in a run time 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. After e' is evaluated, the actor loses its view on that domain. A shared view allows the actor to synchronously invoke read-only methods of all the objects within the corresponding domain. A read-only method is a method that does not modify the state of the domain on which a shared view was acquired. Any attempt to write a field of a domain object while holding only a shared view will result in a run time error. An exclusive view allows the actor to synchronously execute any method on objects belonging to the corresponding domain, regardless of whether they change the state of objects of that domain.

Note that new objects can be created inside shared as well as exclusive views on a domain. As mentioned earlier, the owner domain of a new object is determined based on the lexical scope in which the object is created. By allowing object creation in shared views, it becomes possible to query complex data structures and construct non-trivial query results without requiring an exclusive access to the domain. See Section 5.4 for the semantics of object creation.

4.2. Safety and liveness properties

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

4.2.1. Liveness: Deadlock freedom

The absence of deadlocks and low-level data races are the two properties of the actor model that differentiate it from lower-level concurrency models and provide the useful guarantees to build large concurrent applications safely. 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 4.1, 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, so that all turns take only a finite operational time. Unbounded operations such as infinite loops within a turn are considered programmer errors. They contravene the rules of the language (e.g. SHACL), but are not checked by the implementation. From this follows that all locks held during a turn will be eventually released, which is important to ensure that our language remains deadlock free.

A view on a domain only exists for the duration of a single turn. 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. Again, barring infinite loops while holding an exclusive view.

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 deadlocks cannot be constructed with the basic primitives provided.

As discussed in Section 4.1, views are only held for the duration of a single turn, 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.

4.2.2. Safety: Data race freedom

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

View requests are only allowed on domain references.
All our primitives require that the reference on which a view is requested is a domain reference. Domain objects are not allowed to have direct local references to objects contained in another domain. Instead, objects in other domains have to be referred to with domain references. Without this restriction, multiple actors could share a direct reference to a shared object via different domains, opening the door for classical data races.

Domain references cannot be dereferenced outside of a view.
Each object in the object graph belongs to a certain domain. This means that decomposing an object graph to reveal nested objects will not introduce race condition as access to these nested objects also has to be synchronized by a view.

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 fields and methods As such, any reference to an object that is owned by a domain is always a domain reference.

These three rules ensure that, in any scenario, any domain reference or state that is lexically enclosed by the object to which the domain reference points 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 data races by construction.

4.2.3. Macro-step semantics

The actor model provides an important property for formal reasoning about different program properties. This property is the macro-step semantics [17]. The macro-step semantics says that in an actor model, the granularity of reasoning is at the level of a message/event. This means, each event is processed in a single atomic step, which we have previously referred to as a *turn*. 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. The amount of processing done within a single message can be made as large or as small as necessary, reducing the potential problematic interactions.

After introducing domains and views, the question is whether the macro-step semantics still holds. Arguably, 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 no state is changed.

Based on this reasoning, an actor model with the concept of views presented in Section 4.1 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 [18]. Thus, the original semantics remains preserved and allows the application of relevant reasoning techniques.

4.3. 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 request synchronous access to domain objects protected by views. Listing 5 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 read and update its state in a single turn. 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, domains and views offer a more direct and expressive means to model shared state. Programmers can model shared state without taking into account how this shared state will be accessed from the client side and clients can synchronize and compose access to multiple domain objects in an arbitrary way.

4.4. Summary

In this section we have shown that with the use of views we can avoid the problems discussed in Section 3. Firstly, by *not replicating* the shared state we avoid the need to keep replicas consistent. Secondly, while holding a view we do *not* need to employ *CPS* to access shared state. Because our *when* primitives are asynchronous operations, there is a need to apply CPS when requesting the view. However, once the view is acquired the actor has synchronous access to the domain object and can directly access its fields without using the message-passing system. This means that the actor no longer needs to employ CPS to access the shared object's state. Thirdly, an actor 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 it acquired the view. Lastly, if an actor only uses shared views on a resource it can *read* from that resource *in parallel*.

5. Operational semantics

In this section we describe the operational semantics for a small but significant subset of our language named SHACL-LITE. The operational semantics serves as a reference specification of the semantics of our

language abstractions regarding domains and views. The operational semantics of Shacl-Lite was primarily based on an operational semantics for the AmbientTalk language [19] which in turn was based on that of the Cobox [20] model. Our operational semantics models actors, objects, event loops and domains.

$$
\begin{array}{rcll}
K \subseteq \textbf{Configuration} & ::= & \mathcal{K}\langle A, D, R\rangle & \text{Configurations} \\
a \in A \subseteq \textbf{Actor} & ::= & \mathcal{A}\langle \iota_a, S, E, Q, e\rangle & \text{Actors} \\
d \in D \subseteq \textbf{Domain} & ::= & \mathcal{D}\langle \iota_d, l, O\rangle & \text{Domains} \\
R \subseteq \textbf{Request} & ::= & \mathcal{R}\langle \iota_a, \iota_d, t, e\rangle & \text{View Requests} \\
o \in O \subseteq \textbf{Object} & ::= & \mathcal{O}\langle \iota_o, F, M\rangle & \text{Objects} \\
m \in \textbf{Message} & ::= & \mathcal{M}\langle r, m, \overline{v}\rangle & \text{Messages} \\
n \in N \subseteq \textbf{Notification} & ::= & \mathcal{N}\langle \iota_d, t, e\rangle & \text{Notifications} \\
Q \subseteq \textbf{Queue} & ::= & \overline{m \mid n} & \text{Queues} \\
M \subseteq \textbf{Method} & ::= & \mathrm{m}(\overline{x})\{e\} & \text{Methods} \\
F \subseteq \textbf{Field} & ::= & f : v & \text{Fields} \\
v \in \textbf{Value} & ::= & r \mid t \mid \mathrm{null} & \text{Values} \\
r \in \textbf{Reference} & ::= & \iota_d.\iota_o & \text{References} \\
t \in \textbf{RequestType} & ::= & \textsc{sh} \mid \textsc{ex} & \text{Request Types} \\
l \in \textbf{AccessModifier} & ::= & \textsc{r}(n) \mid \textsc{w} \mid \textsc{f} & \text{Access Modifiers} \\
\iota_d \in S, E \subseteq \textbf{DomainId} & & & \text{Domain Identifiers} \\
i \in \mathbb{N} & & & \text{Integers}
\end{array}
$$

$$\iota_o \in \textbf{ObjectId}, \iota_d \in \textbf{DomainId}, \iota_a \in \textbf{ActorId}, \textbf{ActorId} \subseteq \textbf{DomainId}$$

Figure 7: Semantic entities of Shacl-Lite.

5.1. Semantic entities

Figure 7 lists the different semantic entities of Shacl-Lite. Calligraphic letters like $\mathcal{A}$ and $\mathcal{M}$ are used as "constructors" to distinguish the different semantic entities syntactically instead of using "bare" cartesian products. Actors, domains, and objects each have a distinct address or identity, denoted ι_a, ι_d and ι_o respectively.

In Shacl-Lite a **Configuration** consists of a set of live actors, A, a set of domains, D, and a set of pending view requests, R. A single configuration represents the whole state of a Shacl-Lite program in a single step. In Shacl-Lite each **Actor** has an identity ι_a. Similarly, each **Domain** has an identity ι_d. All actors are associated with a single domain with the same identity as the actor. Domains can also be created by an actor without being associated with that actor. This domain represents the actor's heap. This design decision makes it so that all objects belong to a certain domain and that accessing these objects can be uniformly defined. Because each actor is associated with a single domain (the one with the same Id as the actor, i. e., $\iota_a = \iota_d$), the set of actor Ids is a subset of the set of domain Ids. Each actor has two sets of domain Ids, S and E, on which it currently holds respectively shared and exclusive access. Upon the creation of an actor that actor always has exclusive access to its associated domain. This means that the set E initially contains the singleton identifier ι_a. An actor also has a queue of pending messages Q, and the expression e it is currently evaluating, i. e., reducing. While each actor is associated with a single domain, the inverse does not hold. In addition to an identity, a domain also has a single **Access Modifier** l (or lock) and a set of objects O, that represents the object heap of that domain. A pending **Request** has a reference ι_a, to the actor that placed the request, a reference to the domain ι_d on which the view was requested, the type of request and an expression e, that will be reduced in the context of a view once the domain becomes available. The **Type** of a request is either shared (sh) or exclusive (ex). Note that the way views are assigned to actors implements a simple multiple-reader, single-writer locking strategy. This limits the expressiveness of our implementation in favor of more concise abstractions. An **Object** has an identity ι_o, a set of fields F, and a set of methods M. An asynchronous **Message** holds a reference r, to

the object that was the target of the message, the message identifier m, and a list of values $\overline{v}$, that were passed as arguments. A **Notification** or view is a special type of event that has a reference to the domain on which a view was requested, the type of view that was requested and the expression that is to be reduced once the notification-event is being processed. The **Queue** used by the event loop of an actor is an ordered list of pending messages and notifications. A **Method** has an identifier m, a list of parameters $\overline{x}$, and a body e. A **Field** consists of an identifier f, that is bound to a value v. **Values** can either be a reference r, a request type or `null`.

Shacl-Lite Syntax

Syntax

$$\begin{array}{lll} e \in E \subseteq \textbf{Expression} & ::= & \text{this} \mid x \mid t \mid \text{null} \mid e \,;\, e \mid \lambda x.e \mid e(\overline{e}) \mid \text{let}\, x = e \,\text{in}\, e \mid e.f \mid e.f := e \\ & \mid & e.m(\overline{e}) \mid \text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid \text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \\ & \mid & \text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid e \leftarrow m(\overline{e}) \mid \text{acquire}_t(e)\{e\} \end{array}$$

$x, x_f, x_r \in \textbf{VarName}, f \in \textbf{FieldName}, m \in \textbf{MethodName}$

Runtime Syntax

$$e \quad ::= \quad \ldots \mid r \mid \text{release}_{\text{t}}(\text{v}) \mid \text{object}_{\iota_d}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid r.f : e$$

Evaluation Contexts

$$\begin{array}{lll} e_\square & ::= & \square \mid \text{let}\, x \;=\; e_\square \,\text{in}\, e \mid e_\square.f \mid e_\square.f := e \mid v.f := e_\square \mid e_\square.m(\overline{e}) \mid v.m(\overline{v}, e_\square, \overline{e}) \mid \\ & & e_\square \leftarrow m(\overline{e}) \mid v \leftarrow m(\overline{v}, e_\square, \overline{e}) \mid \text{acquire}_{e_\square}(e)\{e\} \mid \text{acquire}_v(e_\square)\{e\} \end{array}$$

Syntactic Sugar

$$\begin{array}{rcll} e \,;\, e' & \stackrel{\text{def}}{=} & \text{let}\, x = e \,\text{in}\, e' & x \notin \text{FV}(e') \\ \lambda x.e & \stackrel{\text{def}}{=} & \text{let}\, x_{this} = \text{this in} & x_{this} \notin \text{FV}(e) \\ & & \quad \text{object}\,\{ & \\ & & \qquad \text{apply}(x)\{[x_{this}/\text{this}]e\} & \\ & & \quad \} & \\ e(\overline{e}) & \stackrel{\text{def}}{=} & e.\text{apply}(\overline{e}) & \\ \text{whenShared}(e)\{e'\} & \stackrel{\text{def}}{=} & \text{acquire}_{\text{SH}}(e)\{e'\} & \\ \text{whenExclusive}(e)\{e'\} & \stackrel{\text{def}}{=} & \text{acquire}_{\text{EX}}(e)\{e'\} & \end{array}$$

5.2. SHACL-LITE *syntax*

Syntax. SHACL-LITE features both functional as well as object-oriented elements. It has anonymous functions ($\lambda x.e$) and function invocation ($e(\overline{e})$). Local variables can be introduced with a let statement. Objects can be created with the `object` literal syntax. Objects may be lexically nested and are initialized with a number of fields and methods. Those fields can be updated with new values and the object's methods can be called both synchronously ($e.m(\overline{e})$) as well as asynchronously ($e \leftarrow m(\overline{e})$). In the context of a method, the pseudovariable `this` refers to the enclosing object. `this` cannot be used as a parameter name in methods or redefined using `let`.

New domains can be created using the `domain` literal. This creates a new object with the given fields and methods in a fresh domain. New actors can be spawned using the `actor` literal expression. Similarly to domains, this creates a new object with the given fields and methods in a fresh domain. This domain is then

linked with a fresh actor by sharing the same identifier. The newly created actor executes in parallel with the other actors in the system. Expressions contained in domain and actor literals may not refer to lexically enclosing variables, apart from the `this`-pseudovariable. That is, all variables have to be bound except `this`, which means $FV(e) \subseteq \{ \texttt{this} \}$ needs to hold for all field initializer and method body expressions e. Because these expressions do not contain any free variables, actors and domains are isolated from their surrounding lexical scope, making them self-contained.

SHACL's `whenShared` and `whenExclusive` primitives are represented by the $\text{acquire}_e(e)\{e\}$ primitive in SHACL-LITE. The `aquire` primitive is used to acquire views on a domain. It is parametrized with three expressions of which the first two have to reduce to a request type and a domain identifier respectively.

Evaluation contexts. We use evaluation contexts [21] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced. $e_\square$ denotes an expression with a "hole". Each appearance of $e_\square$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

Runtime syntax. Our reduction rules operate on so-called run-time expressions; these are a superset of source-syntax phrases. The additional forms represent references r, the special primitive `release` generated by reducing an acquire statement, object literals that are annotated with the domain identifier of their lexically enclosed domain and field initialisation. This annotation is required so that upon object creation each object gets associated with the appropriate domain.

Syntactic sugar. Anonymous functions are translated to objects with one method named `apply`. Note that the pseudovariable `this` is replaced by a newly introduced variable x_{this} so that `this` still references the surrounding object in the body-expression of that anonymous function. Applying an anonymous function is the same as invoking the method `apply` on the corresponding object.

5.3. Reduction rules

Notation. Actor heaps O are sets of objects. To lookup and extract values from a set O, we use the notation $O = O' \cup \{o\}$. This splits the set O into a singleton set containing the desired object o and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \text{m}$ deconstructs a sequence Q into a subsequence Q' and the last element m. In SHACL-LITE, queues are sequences of messages and notifications and are processed right-to-left, meaning that the last message or notification in the sequence is the first to be processed. We denote both the empty set and the empty sequence using $\emptyset$. The notation $e_\square[e]$ indicates that the expression e is part of a compound expression $e_\square$, and should be reduced first before the compound expression can be reduced further.

Any SHACL-LITE program represented by expression e is run using the initial configuration:

$$\mathcal{K}\langle\{\mathcal{A}\langle\iota_a, \emptyset, \{\iota_a\}, \emptyset, [\![e]\!]_{\iota_a}\rangle\}, \{\mathcal{D}\langle\iota_a, \text{W}, \emptyset\rangle\}, \emptyset\rangle$$

Actor-local reductions. Actors operate by perpetually taking the next message from their message queue, transforming the message into an appropriate expression to evaluate, and then evaluate (reduce) this expression to a value. When the expression is fully reduced, the next message is processed.

If no actor-local reduction rule is applicable to further reduce a reducible expression, i. e., when the reduction is *stuck*, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its message queue is empty, and its current expression is fully reduced to a value. The actor ignores this value and then sits idle until it receives a new message.

We now summarize the actor-local reduction rules in Figure 8:

- LET: a "let"-expression simply substitutes the value of x for v in e.

(LET)

$$\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{let } x = v \text{ in } e]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S, E, Q, e_\Box[[v/x]e]\rangle$$

(PROCESS-MESSAGE)

$$\mathcal{A}\langle \iota_a, S, E, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S, E, Q, \iota_a.\iota_o.m(\overline{v})\rangle$$

(INVOKE)

$$\frac{r = \iota_d.\iota_o \qquad \iota_d \in S \cup E \qquad \mathcal{D}\langle \iota_d, l, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \text{m}(\overline{x})\{e\} \in M}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[r.m(\overline{v})]\rangle\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\}, D, R\rangle}$$

(FIELD-ACCESS)

$$\frac{\iota_d \in S \cup E \qquad \mathcal{D}\langle \iota_d, l, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f : v \in F}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f]\rangle\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D, R\rangle}$$

(FIELD-UPDATE)

$$\frac{\iota_d \in E \qquad o = \mathcal{O}\langle \iota_o, F \uplus \{f : v'\}, M\rangle \qquad o' = \mathcal{O}\langle \iota_o, F \cup \{f : v\}, M\rangle \qquad D = D' \uplus \{\mathcal{D}\langle \iota_d, \text{w}, O \uplus \{o\}\rangle\} \qquad D'' = D' \cup \{\mathcal{D}\langle \iota_d, \text{w}, O \cup \{o'\}\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f := v]\rangle\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D'', R\rangle}$$

(FIELD-INITIALIZE)

$$\frac{D = D' \uplus \{\mathcal{D}\langle \iota_d, \text{w}, O \uplus \{\mathcal{O}\langle \iota_o, F \uplus \{f : v'\}, M\rangle\}\rangle\} \qquad D'' = D' \cup \{\mathcal{D}\langle \iota_d, \text{w}, O \cup \{\mathcal{O}\langle \iota_o, F \cup \{f : v\}, M\rangle\}\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f : v]\rangle\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D'', R\rangle}$$

(CONGRUENCE)

$$\frac{a \rightarrow_a a'}{\mathcal{K}\langle A \uplus \{a\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A \cup \{a'\}, D, R\rangle}$$

Figure 8: Actor-local reduction rules and congruence.

- PROCESS-MESSAGE: this rule describes the processing of incoming asynchronous messages (not for notifications) directed at local objects. A new message can be processed only if two conditions are satisfied: the actor's queue Q is not empty, and its current expression cannot be reduced any further (the expression is a value v).
- INVOKE: a method invocation simply looks up the method m in the receiver object (belonging to some domain) and reduces the method body expression e with appropriate values for the parameters $\overline{x}$ and the pseudovariable `this`. It is *only* possible for an actor to invoke a method on an object within a domain on which that actor currently holds either a shared or exclusive view.
- FIELD-ACCESS, FIELD-UPDATE, FIELD-INITIALIZE: a field update modifies the owning domain's heap so that it contains an object with the same address but with an updated set of fields. Field accesses apply only to objects located in domains on which the actor has either an exclusive or shared view while field updates only applies in the case of an exclusive view. Initializing a field is done by reducing the runtime syntax, $f : e$ and has the same semantics as a field update. A field initialization will always be done right after the object's creation creation and does not require exclusive access to the owning domain.
- CONGRUENCE: this rule simply connects the actor local reduction rules to the global configuration reduction rules.

(NEW-OBJECT)

$$\frac{\iota_o \text{ fresh} \qquad r = \iota_d.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{e'\}}\rangle}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{object}_{\iota_d}\{\overline{f : e}, \overline{m(\overline{x})\{e'\}}\}]\rangle\}, D \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\}, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\overline{r.f : e}; r]\rangle\}, D \cup \{\mathcal{D}\langle \iota_d, l, O \cup \{o\}\rangle\}, R\rangle}$$

(NEW-DOMAIN)

$$\frac{\iota_o, \iota_d \text{ fresh} \qquad o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{[\![e']\!]_{\iota_d}\}}\rangle \qquad r = \iota_d.\iota_o \qquad d = \mathcal{D}\langle \iota_d, \text{F}, \{o\}\rangle}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{domain}\{\overline{f : e}, \overline{m(\overline{x})\{e'\}}\}]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\overline{r.f : [\![e]\!]_{\iota_d}}; r]\rangle\}, D \cup \{d\}, R\rangle}$$

(NEW-ACTOR)

$$\frac{\iota_{a'}, \iota_o \text{ fresh} \qquad r = \iota_{a'}.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{[\![e]\!]_{\iota_{a'}}\}}\rangle \qquad d = \mathcal{D}\langle \iota_{a'}, \text{W}, \{o\}\rangle \qquad a = \mathcal{A}\langle \iota_{a'}, \emptyset, \{\iota_{a'}\}, \emptyset, \overline{r.f : [r/\text{this}][\![e]\!]_{\iota_{a'}}}\rangle}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[r]\rangle, a\}, D \cup \{d\}, R\rangle}$$

Figure 9: Creational rules

Rules for object, domain and actor literals. We summarize the creation reduction rules in Figure 9:

- NEW-OBJECT: All object literals are tagged with the domain id of the lexically enclosed domain. The effect of evaluating an object literal expression is the addition of a new object to the heap of that domain. The fields of the new object are initialized to `null`. The literal expression reduces to a sequence of field initialize expressions. Note that we do not replace the `this` pseudovariable in the field initialize expressions. This means the reference to an object cannot leak before the object is completely initialized. The last expression in the reduced sequence is a domain reference r to the new object.

- NEW-DOMAIN: A domain literal will reduce to the construction of a new domain with a single object in its heap. Similarly to the rule for NEW-OBJECT, the domain literal reduces to a sequence of field initialize expressions. Initially, the domain's access modifier is set to F. After the field initialize expressions are reduced, a domain reference to the newly created object, $\iota_d.\iota_o$, is returned. $[\![e]\!]_{\iota_d}$ denotes a transformation that makes sure that all lexically nested object expressions are annotated with the domain id of the newly created domain. The substitution rules for this are straightforward and are left out of this paper.[5]

- NEW-ACTOR: when an actor ι_a reduces an actor literal expression, a new actor $\iota_{a'}$ is added to the set of actors of the configuration. A newly created domain is associated with that actor. The new domain's heap consists of a single new object ι_o whose fields and methods are described by the literal expression. The domain's access modifier is initialized to W and the domain is added to the set of exclusive domains of the newly created actor. That domain is "owned" by that actor, meaning that no other actors can ever acquire a view on that domain. The $[\![e]\!]_{\iota_d}$ syntax makes sure that all lexically nested object expressions are tagged with the domain id of the newly created domain. As in the rule

[5]Full operational semantics can be found at `http://soft.vub.ac.be/~jdekoste/shacl/operational_semantics`

for NEW-OBJECT, the object's fields are initialized to `null`. The new actor has an empty queue and will, as its first action, initialize the fields of the only object inside its associated domain. The actor literal expression itself reduces to a remote reference to the new object, allowing the creating actor to communicate further with the newly spawned actor.

(LOCAL-ASYNCHRONOUS-SEND)

$$\begin{array}{c}\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S, E, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\square[\text{null}]\rangle\end{array}$$

(DOMAIN-ASYNCHRONOUS-SEND)

$$\frac{\iota_d \notin \textbf{ActorId} \qquad r = \iota_d.\iota_o}{\begin{array}{c}\mathcal{A}\langle \iota_a, S, E, Q, e_\square[r \leftarrow m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{acquire}_{\text{EX}}(r)\{r.m(\overline{v})\}]\rangle\end{array}}$$

(REMOTE-ASYNCHRONOUS-SEND)

$$\frac{\begin{array}{c}A = A' \cup\{\mathcal{A}\langle \iota_{a'}, S', E', Q', e'\rangle\} \\ A'' = A' \cup \{\mathcal{A}\langle \iota_{a'}, S', E', \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}\rangle \cdot Q', e'\rangle\}\end{array}}{\begin{array}{c}\mathcal{K}\langle A \cup\{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle\}, D, R\rangle \\ \rightarrow_k \mathcal{K}\langle A'' \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{null}]\rangle\}, D, R\rangle\end{array}}$$

Figure 10: Asynchronous message rules

Asynchronous communication reductions. We summarize the asynchronous communication reduction rules in Figure 10:

- LOCAL-ASYNCHRONOUS-SEND: an asynchronous message sent to a *local* object (i. e., an object owned by the same actor as the sender) simply appends a new message to the end of the actor's own message queue. The message send itself immediately reduces to `null`.
- DOMAIN-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a domain reference, i. e., a reference whose domain ι_d is not part of the set of **ActorIds**. Reducing an asynchronous message to a domain reference is semantically equivalent to reducing an exclusive view request on that reference and invoking the method synchronously while holding the view (See *View Reductions*). The domain reference is the target of the request and the body of the request is the invocation of the method on that reference. Further reduction of the `acquire` statement will eventually reduce the entire statement to `null`.
- REMOTE-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a remote reference, i. e., a domain reference whose $\iota_{a'}$ is the same as another actor in the system. A new message is appended to the queue of the recipient actor $\iota_{a'}$ (top part of the rule). As in the LOCAL-ASYNCHRONOUS-SEND rule, the message send expression itself evaluates to `null`.

View reductions. We summarize the view reduction rules in Figure 11:

- ACQUIRE-VIEW: This rule describes the reduction of `acquire` expressions. This rule simply adds the view-request to the set of requests in the configuration. Note that this set is not an ordered set and thus requests can in principle be handled in any order. The acquire expression reduces to `null`.
- PROCESS-VIEW-REQUEST: Processing a view request removes that request from the set of requests and updates the access modifier of the domain. How the access modifier is allowed to transition from one value to another is described by the auxiliary function *lock*. Any request to a domain that is currently unavailable will not be matched by *acquire* and cannot be reduced as long as that domain remains

(ACQUIRE-VIEW)

$$\frac{\iota_d \notin \textbf{ActorId}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{acquire}_t(\iota_d.\iota_o)\{e\}]\rangle\}, D, R\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{null}]\rangle\}, D, R \cup \{\mathcal{R}\langle \iota_a, \iota_d, t, e\rangle\}\rangle}$$

(PROCESS-VIEW-REQUEST)

$$\frac{l' = lock(t, l) \\ D = D' \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\} \qquad D'' = D' \uplus \{\mathcal{D}\langle \iota_d, l', O\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e\rangle\}, D, R \uplus \{\mathcal{R}\langle \iota_a, \iota_d, t, e\rangle\}\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, \mathcal{N}\langle \iota_d, t, e\rangle \cdot Q, e\rangle\}, D'', R\rangle}$$

(PROCESS-VIEW-NOTIFICATION)

$$\frac{S', E' = add(\iota_d, t, S, E)}{\mathcal{A}\langle \iota_a, S, E, Q \cdot \mathcal{N}\langle \iota_d, t, e\rangle, v\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S', E', Q, e; \text{release}_t(\iota_d)\rangle}$$

(RELEASE-VIEW)

$$\frac{\iota_d \notin \textbf{ActorId} \qquad l' = unlock(t, l) \qquad S', E' = subtract(\iota_d, t, S, E) \\ D = D' \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\} \\ D'' = D' \cup \{\mathcal{D}\langle \iota_d, l', O\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\square[\text{release}_t(\iota_d)]\rangle\}, D, R\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S', E', Q, e_\square[\text{null}]\rangle\}, D'', R\rangle}$$

Figure 11: Views reduction rules.

unavailable. The auxiliary function *lock* yields the new value for the access modifier given the type of request and the current access modifier of the domain. As a result of processing a request a new notification is scheduled in the requesting actor's queue. Processing a view request can be done in parallel with reducing actor expressions.

- PROCESS-VIEW-NOTIFICATION: Processing a notification will add the domain id, ι_d, to the set of shared or exclusively accessible domains in the actor. Analogous to the processing of messages, a new notification can be processed only if two conditions are satisfied: the actor's queue Q is not empty, and its current expression cannot be reduced any further (the expression is a value v). The actor's set of available shared or exclusive domains is updated according to the request using the *add* function. Processing a notification changes the expression that the actor is currently evaluating to the expression in the notification, followed by a release expression.

- RELEASE-VIEW: Releasing a view on the domain removes that domain id from the set of shared or exclusively accessible domains of the actor. The access modifier of the domain is also updated, potentially allowing other view requests on that domain to be processed. The release statement, which is always the last statement that is reduced by an actor before reducing other messages in its queue, also reduces to `null`.

Auxiliary functions and predicates. The auxiliary function $unlock(t, l)$ describes the transition of the value of an access modifier when releasing it. If a domain was locked for exclusive access its lock will be W (write) and can be changed to F (free). If that resource was locked for shared access we transition either to F or subtract one from the read modifier's value. Similar to the *unlock* rule, the $lock(t, l)$ rule describes the transition of the value of the access modifier of a shared resource when acquiring it. These two rules effectively mimic multiple-reader, single-writer locking.

The *add* and *subtract* rules with four parameters describe the updates to the set of shared, S, and exclusive, E, domain ids of an actor. The *add* rule adds domain ids to both sets while *subtract* rule subtracts domain ids from both sets.

Auxiliary functions and predicates

$$
\begin{array}{rclrcl}
\mathit{lock}(\textsc{ex},\ \textsc{f}) & \stackrel{def}{=} & \textsc{w} & \mathit{unlock}(\textsc{ex},\ \textsc{w}) & \stackrel{def}{=} & \textsc{f} \\
\mathit{lock}(\textsc{sh},\ \textsc{f}) & \stackrel{def}{=} & \textsc{r}(1) & \mathit{unlock}(\textsc{sh},\ \textsc{r}(1)) & \stackrel{def}{=} & \textsc{f} \\
\mathit{lock}(\textsc{sh},\ \textsc{r}(i)) & \stackrel{def}{=} & \textsc{r}(i+1) & \mathit{unlock}(\textsc{sh},\ \textsc{r}(i)) & \stackrel{def}{=} & \textsc{r}(i-1) \\
\\
\mathit{add}(\iota_d, \textsc{ex}, S, E) & \stackrel{def}{=} & S, E \cup \{\iota_d\} & \mathit{subtract}(\iota_d, \textsc{ex}, S, E \mathbin{\dot\cup} \iota_d) & \stackrel{def}{=} & S, E \\
\mathit{add}(\iota_d, \textsc{sh}, S, E) & \stackrel{def}{=} & S \cup \{\iota_d\}, E & \mathit{subtract}(\iota_d, \textsc{sh}, S \mathbin{\dot\cup} \iota_d, E) & \stackrel{def}{=} & S, E
\end{array}
$$

5.4. Object creation in Shacl

In the operational semantics an object is created by initializing its fields to `null` and then reducing a number of field initialization expressions. Object expressions are always lexically nested within an actor or domain and as such, an object can only be created by an actor when that actor has shared or exclusive access to that domain. The reason we do not use field updates for initializing an object is that field updates require the actor to have an exclusive view on the domain of the object. Meaning that, when using field updates, object creation would not be possible in shared mode. This would severely limit the expressiveness of our model. Field initialization expressions of objects are not allowed to use the `this` pseudovariable to refer to the object that is being created. This restriction ensures that object references to an object under construction cannot be leaked to other actors before the object is completely initialized. This restriction is necessary to avoid data races, which can otherwise occur when an object is created inside a domain that was acquired in shared mode. If such an object could be shared with other actors before it is fully initialized, other actors may non-deterministically observe updates to the object as it is being initialized by its creating actor.

In mainstream OO languages such as C# or Java, constructors are allowed to refer to `this`. It is a known problem that this can lead to similar issues, as other objects may then start to interact with objects that are only half-initialized [22].

6. Shacl further features and limitations

Section 4 discussed only the core features of Shacl. In this section we discuss a number of other important features of Shacl as well as a number of drawbacks of our model.

6.1. Further features

6.1.1. 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 on objects in a single domain by multiple actors. A workaround for this problem would be to subdivide the shared data structure into several domains. As an extreme example, we could put each node of a binary search tree 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:

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

The `whenAcquired` primitive takes any two 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 listed domains become available.

SHACL uses *global lock ordering* for all domains. To avoid starvation and deadlocks when views are requested on multiple domains, all such locks should be requested together, instead of one-by-one.

6.1.2. Immutable Domains

SHACL also supports immutable domains for sharing immutable data structures. An immutable domain can be constructed using the `immutableDomain{<expressions>}` syntax. Evaluating this syntax results in the creation of a new read-only domain. The `<expressions>` are evaluated with write permission. This means that any mutation done to the domain has to be done during initialization. In contrast with regular domains, an immutable domain can be referenced without requesting a view. This means that once the immutable domain has been created, all actors can synchronously access any reference to objects within that domain.

Any actor that obtains a domain reference to an object inside an immutable domain can synchronously access that object. Any object created by evaluating `object` literals within the lexical scope of a domain will, similarly to regular domains, also belong to that domain and thus also be immutable after construction. However, a reference to the newly created object cannot be assigned to a field of an object inside that domain. Creating new objects inside an immutable domain can be useful for caching results and broadcasting them to other actors in the system.

6.1.3. Futures

SHACL supports future-type messages. Futures introduce a way for actors to synchronize on the reception of a message. 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. Messages representing *futures* 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 `whenBecomes` expression. 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` expression.

```
1  let cell = object {
2   c : 0;
3   get() {
4     c;
5   }
6   set(n) {
7    this.c := n;
8   }
9  }
```

```
10 let a = actor {
11   increase(counter) {
12     future : counter<-get();
13     whenBecomes(future -> c) {
14       counter<-set(c + 1);
15     }
16   }
17 }
18
19 a<-increase(cell);
```

Figure 12: Illustration of futures

In Figure 12 `get` is sent as a future-type message to the remote reference `counter` and immediately returns a future. 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 3. We still need to employ CPS if we want to access several values of our remote object, as 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 does not return `null` as in SHACL-LITE, but rather a future. 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.

The interactions between views and futures were deliberately left out of the operational semantics for simplicity.

6.2. Limitations: Nesting of Views

Domains and views allow different actors to have shared read access to the object graph of a single domain. If we want our application to support parallel updates of a tree-like structure, we would need to divide the different subtrees of that tree over different domains. This however also implies that while an actor walks down the tree it would need to request new views on the child nodes as it traverses the different domains. One way to do this is trough the use of nested views.

Problematic with the use of nested view requests is that view requests are asynchronous and a nested view is always processed in a later turn, long after the outer view has been processed. Also, one has to take into account that the shared tree may have been modified between releasing the outer view and acquiring the inner view.

```
1  whenShared(parent) {
2    let child = parent.child // child and parent are in different domains
3    whenAcquired([parent], [child]) {
4      if(parent.child == child) {
5        /* success */
6        child.value := 42
7      } else {
8        /* try again */
9      }
10   }
11 }
```

Figure 13: An example of nested views

Figure 13 illustrates this problem. The actor executing the code needs to request a shared view on the domain of the parent node to be able to get a reference on the child node (line 1–2). If that child node is in a different domain from the parent node, that actor will then need to request an exclusive view on the child node to be able to modify it. Because requesting a view is an asynchronous operation, once we acquire the inner view, that view is always processed in a later turn. Thus, we need to re-check if the parent-child relation still holds (line 4). To check this relation we also need read access to the parent domain. We can acquire read access to the parent domain by also requesting a shared view on that domain (line 3). If traversing a tree to a particular leaf, while traversing down the tree, we need to acquire views on each of the nodes on the path to that leaf. On the other hand, one need not worry about deadlocks as all locks are taken and released in a single step. There is no actual nested locking.

The fact that we need to resort to patterns like this to traverse an object graph that is scattered over different domains is an unfortunate but necessary drawback of the fact that view requests are asynchronous operations.

7. Related work

The engineering benefits of semantically coarse-grained synchronization mechanisms in general [23] and the restrictions of the actor model have been recognized by others. In particular the notion of domains

and *view*-like constructs has been proposed before. In this section we will discuss the existing related work by categorizing them into two different categories. On the one hand, there is a body of related work that cares about abstracting away the synchronization of access to shared state with view-like abstractions. On the other hand, there is related work that cares about coarsening the object graphs that are shared with domain-like abstractions.

Related work on view-like abstractions

Demsky views. Closely related to our model are Demsky's and Lam's [23] views, which they propose 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 Demsky and Lam's 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 prohibiting the use of nested views is not enforced by the compiler. Finally, in their approach views are compile-time primitives, which means they cannot be used to safely access shared state depending on runtime information.

Axum. The idea of combining actor-based languages with multiple-reader/single-writer semantics has been investigated previously with the Axum language [24]. The Axum project shares with our work the goal of creating a high level concurrency model that allows structuring interactive and independent components of an application. It 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 defined, which gives some static guarantees about the program but ultimately suffers from the same problems as the views abstractions from Demsky and Lam, especially since Axum provides an explicit escape hatch with the `unsafe` keyword, which allows the language's semantics to be circumvented.

Proactive. ProActive [25] is middleware for Java that provides an actor abstraction on top of threads. It provides the notion of *Coordination actors* to avoid data races 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.

Related work on domains

Ribbons. Another approach similar to our notion of domains is Hoffman's et al. [26] notion of ribbons 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 fewer 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. Access modifiers on protection domains limit the number of

critical operations in which data races need to be considered. But if two threads have write access to the same data structure, access to that data structure still needs to be synchronized.

Deterministic Parallel Java. In Deterministic Parallel Java [27] the programmer has to use effect annotations to determine what parts (*regions*) of the heap a certain method accesses. They ensure data-race-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.

Other related work

Parallel Actor Monitors. The strong restrictions of the actor model with regard to shared state and parallelism have also been discussed earlier. One example are Parallel Actor Monitors [28] (PAM). PAM enables 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 data-structure 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 using PAM.

8. Conclusion

The actor model is a high-level model for concurrent programming. It provides a number of safety guarantees for issues that are often problematic in other models such as deadlock freedom, data-race freedom, and macro-step semantics. Unfortunately the restrictions of this model limit its expressiveness in comparison to less strict implementations such as actor libraries, limiting its suitability for programming shared-memory hardware. The issue of accessing shared state is common to all actor languages. Library-based approaches solve this issue by allowing the programmer to break actor boundaries as an escape hatch (e. g., Scala actors). 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.

Our approach provides a solution specifically for the actor model ensuring maximum interoperability between the different primitives. It introduces the concept of domains. All objects belong to a certain domain and actors can asynchronously request a view on a domain. Once the view is acquired the actor can then synchronously read and/or write to all objects belonging to that domain. In this paper we showed that using domains and views allows for a safe, expressive and efficient sharing of objects between different actors, formalized with an operational semantics for Shacl-Lite. This operational semantics serves as a formal specification of our model. The advantages of this extended event-loop model over traditional actor solutions are threefold. Firstly we partially avoid the continuation-passing style of programming when accessing shared state. Secondly we allow the programmer to introduce extra synchronization constraints on groups of messages and lastly we are able to model true parallel reads. In addition to these benefits, the proposed extension to the actor model still maintains the guarantees for deadlock freedom, data-race freedom, and macro-step semantics.

9. 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.

Stefan Marr is supported by the MobiCra$^{\mathrm{NT}}$ project funded by INNOV$^{\mathrm{IRIS}}$.

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

References

[1] G. A. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, MA, USA, 1986.

[2] J. Armstrong, R. Virding, C. Wikstrom, M. Williams, Concurrent Programming in Erlang, 2nd Edition, Prentice Hall PTR, 1996.

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

[4] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, W. D. Meuter, Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks, in: XXVI International Conference of the Chilean Society of Computer Science (SCCC'07), IEEE Computer Society, 2007, pp. 3–12.

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

[6] S. Srinivasan, A. Mycroft, Kilim: Isolation-typed actors for java, ECOOP 2008–Object-Oriented Programming (2008) 104–128.

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

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

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

[10] Akka, `http://akka.io/`.

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

[12] D. Ungar, R. Smith, Self: The power of simplicity, Vol. 22, ACM, 1987.

[13] E. Brewer, Towards robust distributed systems, in: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, Vol. 19 of PODC'00, 2000, pp. 7–10. `doi:10.1145/343477.343502`.

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

[15] R. E. Johnson, B. Foote, Designing reusable classes, Journal of Object-Oriented Programming 1 (2) (1988) 22–35.
URL `http://www.laputan.org/drc.html`

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

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

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

[19] T. V. Cutsem, C. Scholliers, D. Harnie, An operational semantics of event loop concurrency in ambienttalk, Tech. rep., Vrije Universiteit Brussel (2012).

[20] J. Schäfer, A. Poetzsch-Heffter, Jcobox: Generalizing active objects to concurrent components, ECOOP 2010–Object-Oriented Programming (2010) 275–299.

[21] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical computer science 103 (2) (1992) 235–271.

[22] J. Gil, T. Shragai, Are we ready for a safer construction environment?, in: S. Drossopoulou (Ed.), ECOOP 2009 Object-Oriented Programming, Vol. 5653 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 495–519.

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

[24] Microsoft Corporation, Axum programming language, `http://tinyurl.com/r5e558` (2008-09).

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

[26] K. Hoffman, H. Metzger, 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, ACM, 2011, pp. 289–306.

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

[28] C. Scholliers, É. Tanter, W. D. Meuter, Parallel actor monitors, in: 14th Brazilian Symposium on Programming Languages, 2010.