# Domains: Sharing State in the Communicating Event-Loop Actor Model

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

# Domains: Sharing State in the Communicating Event-Loop Actor Model[☆]

Joeri De Koster[a], Stefan Marr[b], Tom Van Cutsem[a], Theo D'Hondt[a]

[a] *Vrije Universiteit Brussel,*
*Pleinlaan 2,*
*B-1050 Brussels, Belgium*
[b] *Inria Lille,*
*40, avenue Halley,*
*59650 Villeneuve d'Ascq, France*

**Abstract**

The actor model is a message-passing concurrency model that avoids deadlocks and low-level data races by construction. This facilitates concurrent programming, especially in the context of complex interactive applications where modularity, security and fault-tolerance are required. The tradeoff is that the actor model sacrifices expressiveness and safety guarantees with respect to parallel access to shared state.

In this paper we present *domains* as a set of novel language abstractions for safely encapsulating and sharing state within the actor model. We introduce four types of domains, namely immutable, isolated, observable and shared domains that each are tailored to a certain access pattern on that shared state. The domains are characterized with an operational semantics. For each we discuss how the actor model's safety guarantees are upheld even in the presence of conceptually shared state. Furthermore, the proposed language abstractions are evaluated with a case study in Scala comparing them to other synchonisation mechanisms to demonstrate their benefits in deadlock freedom, parallel reads, and enforced isolation.

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

## 1. Introduction

In practice, the actor model is made available either via dedicated programming languages (actor languages), or via libraries in existing languages. Actor languages are mostly *pure*, in the sense that they usually enforce strict 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 [1], SALSA [2], E [3], AmbientTalk [4], and Kilim [5]. 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 downside is that this strict isolation severely restricts the way in which access to shared resources can be expressed.

At the other end of the spectrum, we find actor libraries, which are often built on top of a shared-memory concurrency model. For Java alone, examples include ActorFoundry [6], Actor Architecture [7], ProActive [8], AsyncObjects [9], JavAct [10], Jetlang [11], and AJ [12]. Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [13] and Akka [14]. These libraries have in common that they do not enforce actor isolation, i. e., they cannot guarantee that actors do not share mutable state. There exist techniques to circumvent this issue. For example, it is possible to extend the type system of Scala to guarantee data-race freedom [15].

[☆]This paper builds on De Koster, J.; Marr, S.; DHondt, T. & Van Cutsem, T. (2013) Tanks: Multiple reader, single writer actors. In Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 13, pp. 61-68.

*Email addresses:* `jdekoste@vub.ac.be` (Joeri De Koster), `stefan.marr@inria.fr` (Stefan Marr), `tvcutsem@vub.ac.be` (Tom Van Cutsem), `tjdhondt@vub.ac.be` (Theo D'Hondt)

However, 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. Once the developer choses to go this route, the benefits of the high-level actor model are lost, and the developer typically has to resort to other ad hoc synchronization mechanisms to prevent data races.

A recent study [16] has shown that 56% of the examined Scala programs use actors purely for concurrency in a non-distributed setting. That same study has shown that in 68% of those applications, the programmers mixed actor library constructs with other concurrency mechanisms. When asked for the reason behind this design decision, one of the main motivations programmers brought forward was inadequacies of the actor model, stating that certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state.

In conclusion, pure actor languages are often more strict, which allows them to provide strong safety guarantees. The downside is that they often restrict the expressiveness when it comes to modeling access to a shared resource while impure actor libraries are more flexible at the cost of some of those safety guarantees.

The goal of this work is to enable safe and expressive state sharing among actors in a pure actor language. In this paper, we study the actor model only with the aim of applying it to improve shared-memory concurrency. We do not consider applications that require actors to be physically distributed across machines. In addition, while our approach applies to a wide variety of actor models, in this paper we focus on the Communicating Event-Loop Actor Model (CEL Actor Model) [3] specifically (see Section 2). To achieve this goal, we aim to relax the strictness of the event-loop model via the controlled use of novel language abstractions. We aim to improve state sharing among actors on two levels:

**Safety** The isolation between actors enforces a structure on 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 certain safety and liveness invariants of the "core" application. Thus, as in pure actor languages, we seek 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 seek an actor system in which one can directly express 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, we gain stronger synchronization constraints and prevent the inversion of control that is characteristic for interacting with actors using asynchronous message-passing.

This work extends and unifies the language abstractions presented in De Koster et al. [17] and De Koster et al. [18] in a novel programming model. This unified programming model was formalized in an operational semantics (see Appendix A) and serves as a platform for experimenting with new language abstractions expressing shared state within the actor model. Our validation shows the usefulness of our language abstractions based on a case study in Scala (see Section 5).

This paper is structured as follows. Section 2 introduces the Communicating Event-Loop Actor Model and SHACL [19], a communicating event-loop actor language developed as a platform for experimenting with new language features. Section 3 shows the lack of good abstractions to represent shared resources in modern actor systems by giving an overview of the drawbacks of common techniques to represent shared state in the communicating event-loop model. Section 4 presents the taxonomy that led to the design of four distinct types of domains, namely immutable, isolated, observable, and shared domains. Subsequently each of the different types of domains and their use in SHACL is discussed. Section 5 validates the usefulness of the domain model. We start with a survey of synchronization patterns used by developers in existing open-source Scala projects and then compare these patterns with our domain abstractions. Section 6 discusses related work and Section 7 concludes this paper.

## 2. Shacl: A Communicating Event-Loop Actor Language

*The Communicating Event-Loop (CEL) model.* The CEL model was originally intended as an object-oriented programming model for secure distributed computing [3]. The goal of our Domain Model is to increase the expressiveness of the CEL model in a shared-memory context by allowing safe access to shared state. In the CEL model an actor is represented by a *vat*. Throughout this paper the terms *actor*, *event-loop actor* and *vat* are used interchangeably and always refer to "actors" as defined by the CEL model. As shown in Figure 1, each vat has a single thread of control (the event-loop), a heap of objects, a stack, and an event queue. Each object in a vat's object heap is *owned* by that vat and those objects all share the same event-queue and event-loop. This means that vats are strictly isolated from one another.

Within a vat, references to objects owned by that same vat are called *near references*. Those references can be used as in traditional object-oriented programming to directly invoke methods on the referenced object. Throughout the rest of this paper this is referred to as *sending a synchronous message*. The most distinguishing difference between communicating event-loops and traditional actor models is that traditional actors provide only a single entry point or address for each actor, and thus are first class. In the CEL model vats are not first class entities, i. e. an object cannot hold a reference to a vat. Rather objects can hold references to objects owned by other vats and those references are called *far references*. Such a reference can be used as the target for asynchronous communication. An asynchronous message sent to a far reference is enqueued as an *event* in the event-queue of the actor that owns the target object. Eventually that event will be processed by the event-loop of that vat by forwarding the message to the target object (i. e. directly invoking the method). Processing a single message is called a *turn*. The fact that objects can hold far references to objects owned by another vat does not break actor isolation. Any asynchronous message sent to a far reference will eventually be processed by the owner of the receiver object.

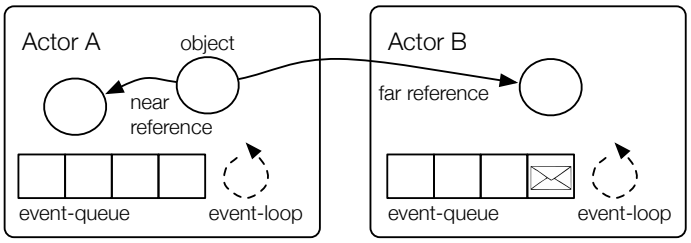


Figure 1: The Domain Model

```
1  a: actor(
2    say_hello():
3      display("Hello World!\n"));
4
5  a<-say_hello();
```

Listing 1: Hello World actor in Shacl

*The* Shacl *Language.* A Shacl VM is started with a single "main actor". All input expressions from the read-eval-print loop (REPL) are evaluated as a single turn of that main actor. Other actors are created by means of the `actor` primitive. Listing 1 illustrates how to create an actor in Shacl. When an actor is created, it is initialized with a new event-loop, object heap and event queue. The event queue is initially empty. The object heap initially hosts a single object which is said to be the actor's *behavior*. In this example, the object defines a single method, namely `say_hello`. The actor that created the new actor, in this case the *main actor*, gets back a *far reference* to that behavior object.

On Line 1, the main actor gets back a far reference to the newly created actor and stores it in the variable `a`. On Line 5, it uses that far reference to send an asynchronous message to the newly created actor.

*Synchronous communication.* The event-loop of an actor can only process synchronous messages (using the dot notation `o.m()`) when the receiver object is owned by the actor that is processing that message. In other words, an actor can only send synchronous messages to near references. Any attempt to synchronously access a far reference is considered to be an erroneous operation and will throw a runtime exception.

*Asynchronous communication.* Actors in Shacl 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 far references to objects owned by another actor (using the arrow notation `o<-m()`). An asynchronous message sent to an object in a different actor is enqueued in the event queue of the actor that owns the receiver object. The thread of execution of that actor is an event-loop that perpetually takes one event (i. e. a queued message send) from its event queue and delivers it to the local receiver object. Hence, events are processed one by one. The processing of a single event is called a *turn*. An important note is that SHACL guarantees message ordering on the outgoing messages towards a single actor. Messages sent from one actor to another will be received in the same order as sent. There are no ordering guarantees on messages sent to multiple distinct actors.

All arguments to an asynchronous message are evaluated before that message is enqueued in the event queue of the receiver actor. Immediate values such as booleans, numbers and strings are sent as a near reference to the receiver actor. Any composite value such as tables, closures or objects are passed by far reference. At the receiver's side there is an extra resolution step to check whether any of the far references are pointing to objects that are owned by the receiver. In this case, the far reference is resolved to a near reference on the receiver side. Note that this is done only on the receiver side, the sending actor still only hold a far reference to that object. In conclusion, any reference to a composite value that is owned by another actor is always a far reference. All other references are always near references.

*The Isolated Turn Principle.*
An important benefit of the semantics of the original actor model is that it enables a *macro-step semantics* [20]. With the macro-step semantics, the actor model provides an important property for formal reasoning about program semantics, which also provides additional guarantees to facilitate application development. The macro-step semantics says that in an actor model, the granularity of reasoning is at the level of a turn, i. e., an actor processing a message from its inbox. This means that a single turn can be regarded as being processed in a single isolated step. Throughout the rest of this paper we refer to this principle as **the isolated turn principle**. The isolated turn principle leads to a convenient reduction of the overall state-space that has to be considered in the process of formal reasoning. Furthermore, this principle is directly beneficial to application programmers, because the amount of processing done within a single turn can be made as large or as small as necessary, which reduces the potential for problematic interactions. In other words, this principle guarantees that, during a single turn, an actor has a consistent view over the whole program environment. Furthermore, the principle guarantees that the actor model, in this case the CEL model, is *free of low-level data races*. However, as the actor model only guarantees isolation within a single turn, high-level race conditions can still occur with bad interleaving of different messages. The general consensus when programming in an actor system is that when an operation spans several messages the programmer must provide a custom synchronization mechanism to prevent bad interleavings and ensure correct execution.

## 3. Representing Shared State in the Communicating Event-Loop Model

If we want to model a shared resource in a pure actor system it must be represented either by replicating the shared resource over the different actors or by encapsulating the shared resource in an additional independent delegate actor. When replicating shared state over the different actors write operations on that shared state typically need to be propagated to the different actors by means of an ad hoc consistency protocol. Additionally, 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. These drawbacks makes replicating the shared state a rarely chosen option among software developers (see Section 5). The most common approach for representing shared state in pure actor systems is by encapsulating the shared state in a separate delegate actor. As a result, any actor that wants to access the shared resource is forced to use asynchronous message passing. There are however four different classes of problems when using this approach: the code is fragmented by the enforced continuation-passing style, read access to the shared resource cannot be parallelized, message-level deadlocks as well as race conditions can occur.

```
1  cell: actor(
2    { content: 0;
3      get(customer):
4        customer<-reply(content);
5      put(new_content):
6        content := new_content });
```

```
7  client: actor(
8    start(cell)
9      cell<-get(
10       object(
11         reply(content):
12           cell<-put(content + 1))));
13
14 client<-start(cell);
```

Listing 2: A shared counter represented by a delegate actor

In this section we discuss these four issues using the example in listing 2. On Line 1 we define a mutable cell as an actor with one field, `content`, and two messages namely `get` and `put`. In this example, we define a client as an actor that first sends a `get` message to retrieve the value of the `cell` and when it receives a reply, sends a `put` message to increment that value in the cell by one.

### *3.1. Code Fragmentation and Continuation-passing Style Enforced*

Using a distinct actor to represent conceptually shared state implies that this resource cannot be accessed directly from any other actor since all communication happens asynchronously within the actor model. Thus, the interface with which to access the shared resource now becomes asynchronous rather than synchronous. This implies the introduction of explicit request-reply-style communication, where the continuation of the request must be turned into a callback. The style of programming where the control of the program is passed around explicitly as a continuation is called continuation-passing style (CPS). The problem with this style of programming is that it leads to "inversion of control" [21].

In the example in listing 2 we see that every time the client wants to retrieve the value of the mutable cell, an extra *customer* parameter needs to be provided. This parameter expects an object that models the continuation of our program provided a callback message, namely `reply`. Consequently, the code that is responsible for incrementing the value of the cell is fragmented over the different callbacks. If that callback would contain another invocation of the `get` message, another callback has to be made. Every time this is done, another level of CPS is introduced.

*Future-type messages.* Many actor languages include future-type messages to overcome this issue of code fragmentation when using callbacks. While typically, asynchronous messages do not have a return value, future-type messages return a future. That future is a placeholder for the return value of the asynchronous message. Once the message is processed, the future is *resolved* with the return value of the message.

```
1 future: cell<-get();
2 future.when_resolved(
3   cell<-put(value + 1));
```

Listing 3: A future-type message in SHACL

In SHACL, every asynchronous message send is a future-type message send and returns a future. Listing 3 illustrates the use of future-type messages in SHACL. We use the return value of the message on Line 1 to store a reference to a future in the `future` variable. Note that the extra `customer` callback parameter has disappeared as this parameter becomes implicit when using futures. Once the `get` message is processed by the actor that owns the `cell` object, that future will be resolved. On Line 2, the `when_resolved` primitive is used to register a closure that needs to be called when the future is resolved.

To maintain the isolated turn principle, two conditions must be met when implementing future-type messages. On the one hand, accessing that future-value has to be an asynchronous operation. Otherwise we would introduce a blocking operation, potentially reintroducing deadlocks. On the other hand, when the future is resolved, the registered closure has to be executed during its own isolated turn. Following from

the isolated turn principle, turns cannot be interleaved, because interleaving would introduce low-level race conditions.

Registering a closure using the `when_resolved` primitive is done explicitly. This forces the programmer to apply CPS. The registered closure then represents the continuation of the program given the return value of the message. Once the future is resolved an event that is responsible for calling the closure is scheduled in the event queue of the actor that issued the request. Thus guaranteeing that the closure is called in its own turn.

Using future-type messages prevents code fragmentation because the callback can be scheduled immediately after sending the message. Additionally, there are techniques to avoid CPS programming by moving the CPS transformation into the compiler (e. g., C# and Scala `assync` and `await` [22, 23]). What is worse is that the operation on our shared resource now spans several turns. Meaning that we can only benefit from the isolated turn principle for each individual turn, but not for the entire operation. Ideally we would like to have synchronous access to the shared resource for the duration of a whole turn to avoid for instance message-level race conditions.

### *3.2. No Parallel Reads*

State that is shared by using a delegate actor can never be read truly in parallel because all accesses to the shared resource are sequentialized by the inbox of the delegate actor. Each request to read (part of) the delegate actor's state is taken out of the inbox and processed one by one. Furthermore, current actor systems do not parallelize reads, even though it would be safe to do so. Other researchers have identified this drawback of the actor model and there exist extensions to the actor model that allow for parallel execution of read-only operations [24].

### *3.3. Message-level Race Conditions*

The traditional actor model does not allow specifying extra synchronization conditions on multiple compound operations. Messages from a single sender are usually processed in the same order as they were sent. However, the order in which messages from different senders are handled is nondeterministic. This means that messages from different senders can be arbitrarily interleaved. Bad interleaving of the different messages can potentially lead to message-level race conditions. In listing 2, the asynchronous messages sent by the `client` on lines 9 and 12 can be interleaved with `get` and `put` messages of other actors, potentially causing a race condition. This type of high-level race condition is typically avoided by increasing the amount of operations in a single turn, i. e., coarsening it up. For example, by introducing an `increment` message that adds a given value to the cell in a single isolated turn. Generally, bad interleaving of messages occurs because different messages, sent by the same actor, cannot always be processed synchronously. Programmers cannot specify synchronization conditions on batches of messages. Therefore, a programmer is limited by the smallest unit of non-interleaved operations provided by the interface of the delegate actor he is using and there are no mechanisms provided to eliminate unwanted interleaving without changing the implementation of the delegate actor, i. e., there are no means for client-side synchronization.

### *3.4. Conclusion*

In most pure actor systems delegate actors are the common idiom to represent a shared resource (replication is rarely used). However, in this section we have shown that using a delegate actor can force computations on that state to span several turns. The guarantees given by the isolated turn principle only apply during a single turn. Common concurrency problems such as deadlocks and race conditions can thus still occur. Unfortunately, using a delegate actor to represent a shared resource forces us to go that route. Ideally, we would want to model our shared resource in such a way that an actor that wants to modify that state can have exclusive, synchronous access to that resource for the duration of a single turn.

## 4. The Domain Model

The goal of the domain model is to increase the expressiveness of the communicating event-loop model by allowing direct synchronous access to shared state while still maintaining the safety guarantees of the CEL model. As we want to change how objects are accessed, the *object heap* is the relevant aspect of the discussion. In the domain model, object heaps are no longer encapsulated by a single actor but are "extracted" from that actor and unified with a different concept called a *domain*. The main difference between a domain and a traditional CEL object heap is that an event-loop can be associated with multiple domains and a domain is not always associated with a single event-loop. However, the various domains are still strictly isolated from one another. How an event-loop can access a domain depends on the domain type and its association with the event-loop. In this section we present the taxonomy that led to the design of four types of domains, namely immutable, isolated, observable, and shared domains. The domain model was formalized in an operational semantics (see Appendix A) which serves as a specification for the precise semantics of our model.

### *4.1. Taxonomy*

Rather than taking the fine-grained approach of allowing to synchronize access to individual objects, with the domain model we take a more coarse-grained approach by synchronising access to whole object heaps (i. e. domains). How the different event-loops can access those domains is restricted along two axes. Firstly, the number of event-loops that have synchronous read and write access to a domain is considered. For any given execution, let $RW(d)$ denote the set of event-loops that are allowed to synchronously read from and write to objects owned by a domain $d$. If $N$ is the number of event-loops during this execution, it follows that $0 <= |RW(d)| <= N$ for any given domain $d$. For a disciplined concurrency model, the relevant cases are that either 0, 1 or $N$ event-loops have synchronous write access to the same domain $d$, and thus, the design space is constrained by:

$$\forall d : |RW(d)| \in \{0, 1, N\} \quad (1)$$

The main notion is that for $|RW(d)| = 0$ the domain $d$ is read-only, meaning that no event-loop can ever have synchronous write access to that domain. For $|RW(d)| = 1$, there is a single event-loop that has synchronous write access to the domain. We call this event-loop the *owner* of that domain. For $|RW(d)| = N$, all event-loops in the system can potentially read from and write to domain $d$. However, that does not necessarily imply *undisciplined* simultaneous writes, instead it means that any event-loop can at some point during the execution have synchronous write access to objects owned by that domain. Similarly, let $R(d)$ denote the set of event-loops that have synchronous read-only access (and no synchronous write access) to objects owned by domain $d$. This implies that $RW(d)$ and $R(d)$ are disjoint sets ($RW(d) \cap R(d) = \emptyset$). This means that $|RW(d) \cup R(d)|$ is the number of event-loops that have some form of access to domain $d$, whether it be read-only or read-write access. It follows that $0 <= |RW(d) \cup R(d)| <= N$. If $|RW(d) \cup R(d)| = 0$ then the domain is inaccessible and therefore useless. For a disciplined concurrency model, the relevant cases are that either 1 or $N$ event-loops can have synchronous access to a domain. This leads us to the following formula:

$$\forall d : |RW(d) \cup R(d)| \in \{1, N\} \quad (2)$$

If $|RW(d) \cup R(d)| = 1$ then only a single event-loop can synchronously access objects inside that domain. In the domain model, that event-loop is said to be the *owner* of that domain. If $|RW(d) \cup R(d)| = N$, then every event-loop can have synchronous access to the domain.

Based on the design space resulting from combining the formulas (1) and (2), we identify five useful settings for domains and Table 1 names them in relation to the design space.

Note that, for completeness, *immutable isolated domains* are included in this table. However, we do not consider them in SHACL. There is no point in having a domain that is both immutable and isolated because they are subsumed under immutable domains where the creator of the domain does not expose any reference to objects inside that domain.

| $\|RW(d)\|$ | 0 | | 1 | | $N$ |
|---|---|---|---|---|---|
| $\|R(d)\|$ | 1 | $N$ | 0 | $N-1$ | 0 |
| | ***immutable isolated*** | ***immutable*** | ***isolated*** | ***observable*** | ***shared*** |

Table 1: The different types of domains

### *4.2. Immutable, Isolated, Observable and Shared Domains*

An instance of each of the domains presented in Section 4.1 is represented by the dotted boxes in Figure 2.

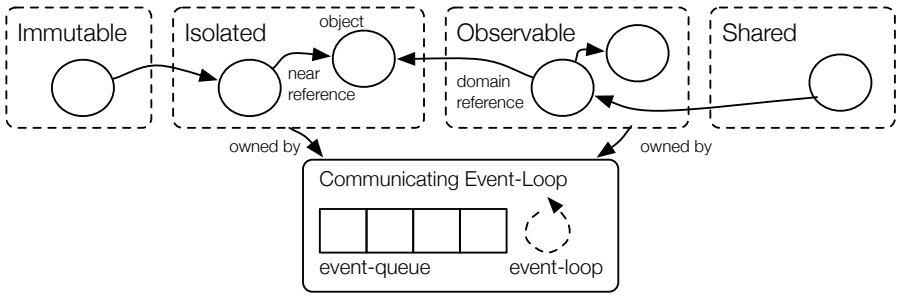


Figure 2: The Domain Model

Similarly to vats, domains are not first class software entities (i. e., an object can never hold a reference to a domain). Rather objects can hold references to objects owned by other domains and those references are called *domain references*. How a domain reference can be accessed depends on the type of the domain that owns the target object and its relation to the event-loop that is accessing that reference.

In Figure 2, objects are shown as circles and references between different objects are shown as arrows. Also, the isolated domain and the observable domain have an arrow to illustrate the ownership relation between the domain and the shown event-loop. Within a domain, references to objects owned by the same domain are called *near references* and have the same properties as in the original model. Namely, near references can be used for synchronous communication. References to objects within a different domain are called *domain references*. A domain reference is always typed with the type of domain that owns the referred object, e. g., a reference to an object owned by an isolated domain is called an *isolated domain reference*. Furthermore derived from the CEL model, the following general notions apply:

*Synchronous or asynchronous communication.* The type of domain reference does not restrict whether that reference is held exclusively by one actor or shared by multiple actors. Any reference can be shared by an event-loop with other event-loops by sending it via message passing. Rather, the type of domain determines whether the referenced object can be synchronously or asynchronously accessed by a particular event-loop. This is the same for references in the original communicating event-loop actor model. Whether references are synchronously accessible in the original model is determined by the type of reference (near or far).

*Lexical ownership rule.* Similar to the original model, any lexically nested object expressions always evaluate to objects that are owned by the lexically enclosing domain. Those objects can have direct references to one another, called *near references*, but can also obtain references to objects in other domains. Such references are called *domain references*. Because of this distinction between both types of references, the domain model guarantees that different domains are strictly isolated from one another.

*Isolated turn principle.* Similar to the original model, the isolated turn principle still applies. By allowing concurrent synchronous access to domains we can potentially have parallel execution of certain operations on objects within the same isolated heap. This also means that, depending on the way those objects are accessed, we need to introduce some synchronization mechanism to guarantee the isolated turn principle.

The required synchronization mechanism and its performance characteristics highly depend on the type of the domain and its concrete realization.

*Asynchronous communication with a domain reference.* The general rule is that, if the domain has an owner then the asynchronous message is always enqueued in the event queue of the event-loop that owns the domain. If however the domain does not have an owner, the message is enqueued in the event queue of the sender of the message. The rationale here is to stay faithful to the original model. In the original model an asynchronous message is always enqueued in the event queue of the owner of an object. In the case that a domain does not have a particular owner, the message is enqueued in the senders own event queue.

*Synchronous communication with a domain reference.* The general rule is that the owner of a domain has fully unrestricted synchronous access to the domain. Whether or not other event-loops can synchronously invoke methods on a domain reference wholly depends on the type of domain reference. The following sections illustrate for each domain type what event-loops can synchronously access objects owned by those domains.

*An Operational Semantics for* SHACL-LITE.
In Appendix A we provide a small step operational semantics for a small but significant subset of SHACL, named SHACL-LITE. The aim of this operational semantics is to serve as a formal specification of the domain model. Because of space constraints, the full operational semantics is only part of the appendix. In the remainder of this and the following sections, we sketch the main ideas used to construct the semantics.

The operational semantics of SHACL-LITE is primarily based on an operational semantics for the AmbientTalk language [25] which in turn is based on the Cobox [26] model. Hence, the basic operational semantics models a communicating event-loop actor language featuring anonymous functions, objects, method invocation, field access and actors. The object heap of an actor is represented by a an isolated domain, which is semantically equivalent to a CEL vat. Since object heaps are separated from actors, we can extend the basic semantics subsequently with additional domains with minimal changes to the existing rules. It also allows for a uniform definition of object ownership.

### *4.2.1. Immutable domains*

An immutable domain represents an object heap of immutable objects. Immutable domains are useful when different actors need to share immutable objects, for example, when sharing library code. A reference to an object owned by an immutable domain is called an immutable domain reference. Any event-loop can use such a reference to synchronously invoke methods and read fields of the referenced object. Writing to a field of an object that is owned by an immutable domain will result in a run-time error.

$$\forall d \in Immutable : RW(d) = \emptyset \wedge |R(d)| = N$$

SHACL's immutable domains are created with the `immutable` primitive. Figure 3 shows an example using an immutable domain. On Line 1, the main event-loop creates a new immutable domain using the `immutable` primitive. That newly created immutable domain is initialized with a single object with one field, `pi` and one method `circle_area`. Invoking the `immutable` primitive returns an *immutable domain reference* to the object. In our example the reference is stored in the variable `formulas` and then used on Line 7 to synchronously invoke the `circle_area` method. Any event-loop that obtains an immutable domain reference to the object is allowed to **synchronously** invoke methods on that object as long as that method is a read-only method, i. e., it does not modify the state of objects in the domain. Creating a new object inside an immutable domain is allowed because that operation does not mutate any fields. That object will also be allocated on the heap of the immutable domain. By allocating that object on the heap of the immutable domain all rules for immutability also apply to that object. It's uncommon practice to send asynchronous messages to an immutable domain reference because synchronous access is always allowed. However, when an actor sends an **asynchronous message** to an immutable domain reference, that message is enqueued in the sender's own event queue.

```
formulas: immutable(
  { pi: 3.14;
    circle_area(r):
      pi * r * r });

r: 5;
A: formulas.circle_area(r);
```

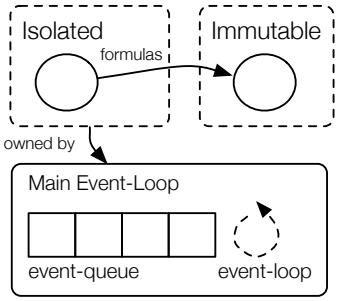


Figure 3: An immutable domain defining some constants

*Discussion.*
Note that an application does not necessarily benefit from having multiple immutable domains. While it would be sufficient to have one global immutable domain per application the immutable domain syntax here serves as a lexical demarcation of which objects are immutable and which are not. Any object instantiated from an object expression that is lexically nested in an immutable domain is immutable. That object can be synchronously accessed from within any actor that obtains a reference to that object.

*Properties.*
That the **isolated turn principle** still holds for immutable domains can be trivially shown. During a turn of an event-loop that event-loop is guaranteed to have a consistent view of all objects in the whole domain as all of those objects are immutable. Furthermore, since no additional operations are introduced, **deadlock freedom** is not affected either.

*Semantics.*
The semantics for immutable domains in Appendix A.3 introduce an immutable domain type with its associated identifier and object heap. By not specifying a field-update rule for immutable domains it is impossible to reduce field updates on an immutable object, which is semantically equivalent to a runtime error. Synchronous method invocation (IMMUTABLE-INVOKE) and field access (IMMUTABLE-FIELD-ACCESS) are defined and are reduced as if the immutable domain reference was a near reference. Object expressions that are lexically enclosed by the immutable domain will be tagged with the identifier of that domain (NEW-IMMUTABLE-DOMAIN). Any object expression that is tagged with an immutable domain identifier will reduce to an immutable domain reference to an object that is added to the immutable domain (NEW-IMMUTABLE-OBJECT). Lastly, any asynchronous message that is sent to an immutable domain reference is enqueued in the sender's own event queue. This is evidenced by the IMMUTABLE-ASYNCHRONOUS-SEND rule.

### *4.2.2. Isolated domains*

An isolated domain is the equivalent of an object heap in the original communicating event-loop actor model. In the original model, only the "owner" of the objects could synchronously read from and write to those objects. Similarly, only one event-loop can synchronously read from and write to *isolated domain references*, namely the owner of the isolated domain. Domains are a generalization of the object heaps in the original communicating event-loop model. Every newly created actor is associated with its own event queue, event-loop and isolated domain. Such isolated domains share the same use cases as communicating event-loops, i. e., they are ideal to represent coarse-grained subsystems that do not need to share their internal state and benefit from the strong consistency properties.

$$\forall d \in Isolated : |RW(d)| = 1 \wedge R(d) = \emptyset$$

The initial configuration of the SHACL VM consists of a *main event-loop* and its associated isolated domain. The main program text is executed by the main event-loop with respect to that domain (i. e. any new objects will be created inside the main isolated domain). Spawning a new actor using the `actor` primitive creates a new event-loop together with its associated isolated domain. Figure 4 illustrates how an isolated domain is instantiated when creating a new actor. On Line 1 the main event-loop invokes the `actor` primitive. A new communicating event-loop will be created together with its own isolated domain in place. An isolated domain is always owned by a single event-loop, in this case the newly created one. The domain is initially empty but for a single object that is initialized from the call-by-name parameter of the `actor` primitive. The return value of the actor primitive is an *isolated domain reference* to that object, which is stored in the variable `a`.

```
1 a: actor(
2   say_hello():
3     display("Hello World!\n"));
4
5 a<-say_hello();
```

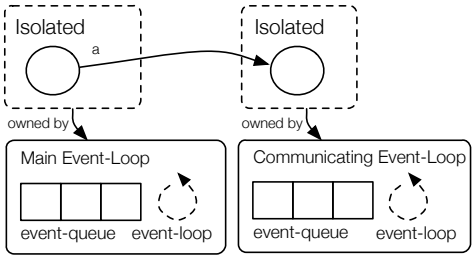


Figure 4: Two communicating event-loops and their isolated domains.

From the perspective of the owner of the isolated domain, isolated domain references are the same as *near references* of the original model. The event-loop that owns the isolated domain can **synchronously** access that reference. From the perspective of other event-loops, an isolated domain reference is the same as a *far reference* of the original model. Any event-loop that obtained a reference to the isolated object has to employ asynchronous communication to access the object. In our example, the main actor has to send an asynchronous `say_hello` message on Line 5. That message will be enqueued in the event queue of the owner of the domain. Any attempt to synchronously access an far reference by an event-loop other than the owner of the domain is considered to be an erroneous operation and will result in a runtime error in SHACL.

*Properties.*
To mimic the semantics of traditional event-loop actors, SHACL actors are created together with their own isolated domain in place. That means that all objects that are created from lexically nested `object` expressions will belong to the isolated domain of the actor. Because actors can only synchronously access objects from their own isolated domain, actors and their state are still fully isolated from one another. That also means that isolated domains preserve the same properties with regard to **isolated turns** and **deadlock freedom**.

*Semantics.* The semantics of isolated domains are included in the initial operational semantics for SHACL-LITE as defined in Appendix A.2. Each isolated domain has an owner that has the same identifier as an actor in the configuration. Method invocation (INVOKE) as well as field access (FIELD-ACCESS) and update (FIELD-UPDATE) rules can only be further reduced if the actor that is accessing the isolated domain is also the owner of that domain. If this is not the case the expression cannot be further reduced which is semantically equivalent to a runtime error.

### *4.2.3. Observable domains*

The motivation behind observable domains is to allow the programmer to express shared state that belongs to (i. e. can be synchronously read and updated by) a single event-loop but is synchronously observable by others. For example, in an MVC application, a model actor could define an observable domain for objects to be observed (but not modified) by different view actors. Similar to other domains, an observable domain

is a container for observable objects. In the context of the domain model, an observable object is an object that is synchronously readable by all event-loops in the system.[1] Furthermore, it has a single owner that is allowed to synchronously read from and write to objects that belong to that domain. All other event-loops that have obtained an observable domain reference are allowed to use that reference to synchronously invoke methods. However, that method has to be read-only, an event-loop is not allowed to modify a field of an observable object owned by another event-loop.

$$\forall d \in \mathit{Observable} : |RW(d)| = 1 \wedge |R(d)| = N - 1$$

Figure 5 illustrates the use of observable domains in SHACL. On Line 1, the main event-loop creates a new observable domain. That domain is initialized with a single object that defines a `get` and a `set` method. By creating the observable domain, the main event-loop becomes the owner of that domain. The return value of the `observable` primitive is an *observable domain reference* that is stored in the `counter` variable. On Line 8, the main event-loop creates a new actor. The isolated domain reference that is returned from the `actor` primitive is stored in the `observer` variable. On Line 12, the main event-loop increases the value of the counter synchronously. The main event-loop is allowed to do that because it is the owner of the domain. On Line 13, the main event-loop sends an asynchronous `increase` message to the observer actor passing the observable domain reference to the counter as an argument. On Line 10, the observer actor can synchronously read from the observable domain reference but has to send an asynchronous message to write to that reference. Any attempt by a non-owner event-loop to synchronously write to a field of an observable domain object will result in a runtime error. The asynchronous `set` message sent to the counter on Line 10 is enqueued in the event queue of the owner of the observable domain, in this case the main event-loop.

```
1  counter: observable(
2    { val: 0;
3      set(v):
4        val:= v;
5      get():
6        val });
7
8  observer: actor(
9    increase(counter):
10     counter<-set(counter.get() + 1));
11
12 counter.set(counter.get() + 1);
13 observer<-increase(counter);
```

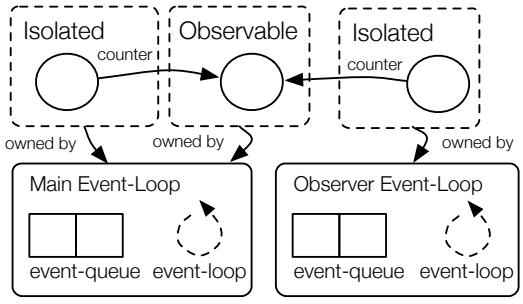


Figure 5: An observable domain owned by an event-loop

*Observable Actors.*
Actors in SHACL are initialized with an event-loop and an isolated domain. It is possible to envision actors that, upon creation, are associated with an observable domain. This approach was explored in earlier work [17]. In that case, the whole state of the actor becomes observable and any shared reference will be synchronously readable by any other event-loop. However, currently in SHACL, we chose to associate actors with an isolated domain by default because of the similarities with the object heap of a vat in the original model.

*Properties.*
The execution of the program in Figure 5 consists of three turns. Firstly, there is the main event-loop that is executing the program text in a single turn (Line 1 to Line 13). Secondly, there is the turn of the observer

[1] Not to be confused with `java.util.Observable` or the Observer design pattern.

actor that processes the `increase` message (Line 10). Thirdly, there is the turn of the main event-loop that has to process the `set` message sent by the observer actor (Line 3). If the isolated turn principle is to be preserved, during all of these turns the event-loops have to have a consistent view over all synchronously accessible state. For example, the observer actor can read from the counter several times and will always observe the same value, regardless of whether the counter is being updated concurrently by the main event-loop. Conceptually, the observer first takes a snapshot of the whole observer domain before executing the `increase` method. For the duration of its turn, the observer always observes the same value from the `counter`, even when that counter is being modified concurrently. From the perspective of the owner of the domain, the turn also has to be isolated. That means that any update to the observable domain should only be visible by other event-loops at the end of the turn. In our example this has some implications on what value is observed by the observer actor at the start of its turn. Either the main event-loop has not yet finished its first turn and the observed value will be `0`. Or the main event-loop has finished its turn (and is now idle) and the observed value will be `1`. Either way, once the turn of the observer actor starts it will always see the same value for each invocation of the `get` message regardless of any updates of the main event-loop. This exposes a race condition on the event level because the value passed to the asynchronous `set` message on Line 10 might be based on an outdated value. Observable domains offer a means to uncoordinated reads of shared state in a safe way but any asynchronous writes still need to be coordinated in some way to avoid race conditions; which is in this case viable because the isolated turn principle only guarantees isolation during a single turn. Note that, in this simple example, the observable domain only contains a single object. Per the lexical ownership rule for domains, any nested `object` primitives will evaluate to objects that are also owned by the same observable domain. During a single turn, any event-loop always has a consistent view over the whole observable domain.

*Semantics.*
The semantics for observable domains are introduced in Appendix A.4. The PROCESS-MESSAGE rule is adapted to ensure that before starting the turn and further reducing the message, an actor takes a snapshot of each observable domain's object heap. The OBSERVABLE-FIELD-ACCESS rule is defined in such a way that it always reads the value of a field in that local snapshot. The OBSERVABLE-FIELD-UPDATE rule is similarly defined with the only exception that only the owner is allowed to modify a field in its local snapshot. The COMMIT rule specifies that at the end of each turn an actor will always replace the object heap of an observable domain with its own local snapshot. This means that changes to the object heap of an observable domain only become visible at the end of each turn of the owner of that domain.

*A Note on the Implementation.*
To guarantee the isolated turn principle, SHACL uses a specialized version of Software Transactional Memory [27]. Other implementation strategies may be used to ensure consistency. However, for SHACL, we use transactional memory to ensure isolation for the processing of events that use observable domains. The processing of events has a transactional behavior, as such, STM is a good implementation method for observable domains. However, it is important to note that while events have transactional behavior, the SHACL model does not have any STM specific keywords to delineate transactions. Rather than introducing new keywords, the processing of a single event is considered a single transaction and observable domain references are considered to be references to transactional memory. During a turn an event-loop can execute non-idempotent operations such as side-effects on non-transactional memory and I/O operations. As such, a conventional STM model would not be suitable to implement observable domains. To ensure that all events are processed only once the STM should avoid aborting transactions. Not being able to support aborting transactions means we have the following two restrictions for our STM. On the one hand, *all writers have to have access to the latest version of the memory.* Writing to a memory location based on an old version would otherwise cause the transaction to be in an inconsistent state and would have to be aborted. On the other hand, *all readers have to see values from a single snapshot of the memory.* Otherwise readers could read from a memory location that changed value during a transaction which would then need to be aborted. A Multi-Version History STM [28] is an optimistic approach to transactional memory where read-only transactions are guaranteed to successfully commit by keeping multiple versions of the transactional

objects. The only transactions that can abort in such a system are conflicting writers. An observable domain allows only a single writer for each object, namely the owner of that domain. If there is only one writer, there cannot be any conflicting writers. If all readers are guaranteed to succeed and there are no conflicting writers, all transactions will succeed and we have successfully supported both restrictions.

#### *4.2.4. Shared domains*

The motivation behind shared domains is to allow the programmer to express shared state that does not belong to a particular event-loop but is rather shared among multiple event-loops. A shared domain allows any event-loop to synchronously read from and write to objects belonging to that domain. A shared domain does not have a particular owner but event-loops do have to obtain read or write access rights before being able to access a *shared domain reference*.

$$\forall d \in Shared : |RW(d)| = N \wedge R(d) = \emptyset$$

Any event-loop can **synchronously access** shared domain references for as long as they have a so-called *view* on that domain. A view can be acquired using either the `when_exclusive` or the `when_shared` primitive.

```
1 counter: shared(
2   { val: 0;
3     set(v):
4       val:= v;
5     get():
6       val });
7
8 counter.when_exclusive(
9   counter.set(counter.get() + 1));
```

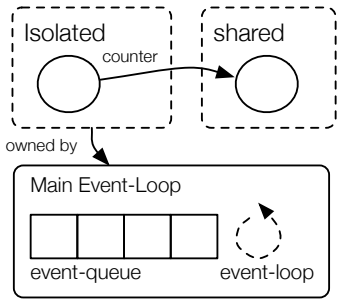


Figure 6: A shared domain

Figure 6 illustrates the usage of shared domains. On Line 1 the main event-loop creates a shared domain. Similar to other domain primitives, the shared domain is initialized with a single object created from the call-by-name parameter of the `shared` primitive. The result of invoking the `shared` primitive is a *shared domain reference* to that object. In our example that reference is stored in the `counter` variable. Any attempt to synchronously invoke a method on such a reference outside of a view is considered to be an erroneous operation and will result in a runtime error. On Line 8, an exclusive view is requested on the domain using the `when_exclusive` primitive. The request is stored in a *view-scheduler* and immediately returns (i. e. the view primitives are asynchronous operations). The main event-loop continues and ends its current turn. A shared domain is available for exclusive access when it is not locked by any other event-loop for shared or exclusive access. When the shared domain becomes available for exclusive access two things happen. Firstly, the domain is locked for exclusive access. Secondly, an event that is responsible for evaluating the call-by-name parameter of the `when_exclusive` primitive is put in the event queue of the main event-loop. This event is called a *view*. Because of this, views are only processed between two turns of the event-loop that requested the view. While that view is being processed, the main event-loop has exclusive synchronous access to the shared counter domain. The evaluation of the view is considered to be a single turn and once that turn ends, the shared domain is freed again, allowing other event-loops to access it.

Note that, in this simple example, the shared domain only contains a single object. Per the lexical ownership rule for domains, any nested `object` primitives will evaluate to objects that are also owned by the same shared domain. During a view, the event-loop that requested the view always has a consistent view over the whole shared domain, not only the object for which the view was requested. A shared view

can be requested by using the `when_shared` primitive. Requesting a shared view is also an asynchronous operation. When the domain becomes available for shared access a view is scheduled in the event queue of the event-loop that issued the request. A domain is available for shared access when it is not locked for exclusive access. Multiple shared views on the same domain, requested by different event-loops, can exist simultaneously. During a shared view the event-loop has synchronous read-only access to all objects in the shared domain to which it holds a reference. Any attempt to modify a field of a shared object during a shared view is considered to be an erroneous operation and will result in a runtime error. Note that the view expression has access to its surrounding lexical scope. A view is always executed by the event-loop that requested the view and during that view the event-loop has synchronous access to both its own isolated domain as well as the shared domain.

*Properties.*
Shared domains require that event-loops obtain a *view* to interact with the contained objects. The view-scheduler guarantees that no conflicting views are scheduled at the same time, which ensures that the **isolated turn principle** is maintained. The view-scheduler prioritizes exclusive views to prevent starvation of exclusive view requests by repeated shared view requests and to ensure **fair scheduling** of the different views. Event-loops that request a shared view have to wait until all exclusive view requests have been handled, even if a shared view was already granted to a different event-loop. Because all newly introduced view primitives are non-blocking, **deadlock freedom** is also guaranteed. The implementation employs a global lock ordering technique to ensure that all domains can be locked at the same time in case of a `when_acquired`.

*Semantics.*
The semantics for shared domains are introduced in Appendix A.5. The rules for accessing an object owned by a shared domain are defined in such a way that only actors that have acquired a shared or exclusive view can invoke methods (SHARED-INVOKE) on or read from fields of a shared object (SHARED-FIELD-ACCESS). Updating a field is restricted in such a way that such an expression can only be reduced for actors that have acquired an exclusive view on the domain (SHARED-FIELD-UPDATE). A new primitive is added to request a view on a shared domain (ACQUIRE-VIEW). Requesting a view is simply reduced to adding that request to the unordered set of requests for that domain. In the semantics, requests are handled in a random order and reducing a request locks the domain for shared or exclusive access and adds a notification to the queue of actor that originally initiated the request (PROCESS-VIEW-REQUEST). A notification is a new type of event during which the actor that initiated the request has synchronous read-only or read/write access to the domain (PROCESS-VIEW-NOTIFICATION). After the event is processed, the domain is released again by releasing the lock, after which other requests on the same domain can potentially be further reduced (RELEASE-VIEW).

### *4.3. Benefits of Domains*

In the case of *pure actor systems* the more common solution to representing a shared resource is to encapsulate that shared resource in a delegate actor. Using domains has a number of distinct advantages over this approach.

*Message-level Race Conditions.*
Isolation of any number of operations is only guaranteed during a single turn (because of the isolated turn principle). Using a delegate actor requires the client side to send an asynchronous message upon each access of the shared state. Because each asynchronous message is processed in its own turn, the client loses the benefit of the isolated turn principle when combining different messages. In other words, the client side is unable to put extra synchronization conditions on batches of messages. Note that message-level race conditions still exist in SHACL. However, using a domain allows the client to synchronously access objects owned by that domain multiple times over the course of a single turn, effectively allowing the programmer to combine different operations on the domain objects in a larger synchronous operation.

*Parallel Reads.*
The issue with a delegate actor representing shared state was that all of the operations on that shared state were serialized by the event queue of that actor. The immutable, observable and shared domains can all have many readers and each of those domains allows all readers to synchronously read from objects owned by those domains in parallel.

*Continuation Passing Style.*
Asynchronous operations always force programmers to apply CPS to the surrounding code. The domain abstractions allows programmers to transform a number of asynchronous operations into synchronous operations effectively avoiding a CPS transformation in those places. One exception is a shared domain where one level of CPS is still required, namely to request the view on the domain. However, during that view, the event-loop has unlimited synchronous access to any object owned by the shared domain without requiring a CPS transformation.

*Access to Its Own Isolated Domain.*
During any given turn, if an event-loop in SHACL has synchronous access to a domain that event-loop can also still synchronously access its own isolated domain. This means that an event-loop can combine access to its own state with access to the domain during that turn. This can be beneficial when the update to the domain depends on state that is local to the event-loop. When using a delegate actor, this is impossible because different event-loops are strictly isolated from one another.

## 5. Validation: A Case Study in Scala

The goal of our case study was to identify the software patterns used by developers to model access to a shared resource in the actor model. We then show that these patterns can be converted to an alternative implementation using domains and compare the properties of the different approaches. For our case study we use the corpus of Tasharofi et al. [16] which is publicly available online.[2] From the initial set of around 750 Scala programs available on GitHub,[3] 16 real-world projects were selected based on the following three criteria: Firstly, the project has to use either the Scala or Akka actor library to implement a portion of its functionality. Secondly, the project must consist of at least 3000 lines of code combined over Scala and Java. Lastly, the project had to be actively maintained by at least two developers.

During the case study we identified three distinct patterns that programmers currently use to synchronize access to a shared resource, namely client-side locks, server-side locks and delegate actors. In this case, the server side is the actor hosting the shared resource and the client side are the actors that require access to the shared resource. In the entire corpus we found 232 code examples where developers represent a shared resource and classified them according to these three patterns. The most common pattern (166 occurrences) was the use of a server-side lock where synchronized access to the shared resource is usually done by acquiring the same lock each time the shared resource is accessed. The main advantage of this approach is that synchronization is enforced upon each client-side access of the shared resource. Another pattern that was identified is the use of a lock on the client side (38 occurrences) for synchronizing access to the shared resource. The main benefit of this approach over a server-side lock is that the client using the shared resource can compose multiple operations in a larger synchronous operation. The downside is that synchronization is not enforced and clients that do not acquire the lock before accessing the shared resource can cause race-conditions to occur. A third pattern is the use of a delegate actor (28 occurrences) to encapsulate and synchronize access to the shared resource. The main advantage of using a delegate actor is that deadlock freedom is guaranteed.

Next we give an overview of a number of desirable properties for any synchronization mechanism. The goal of our validation is to evaluate the different synchronization patterns found in the survey according to these properties.

[2] http://actor-applications.cs.illinois.edu/index.html
[3] https://github.com

- **No client-side CPS**. When the employed synchronization mechanism is non-blocking, then the client side code typically needs to employ an event-driven style where the code is structured in a continuation passing style. Since this leads to code fragmentation, a direct style is preferable.
- **Deadlock free**. Blocking synchronization mechanisms do not suffer from the CPS issue as any access to the shared resource will block until it yields a result. However, blocking synchronization mechanisms can potentially introduce deadlocks when nested while non-blocking synchronization mechanisms are usually deadlock-free.
- **Parallel reads**. Multiple read-only operations can be trivially parallelized without introducing race conditions. However, making the distinction between read and write operations often comes with a cost and as such, not all synchronization mechanisms include this optimization.
- **Enforced Synchronization**. If the synchronization mechanism is put on the server side, then the server side can typically enforce synchronization for any client-side access of the shared resource.
- **Composable Operations**. If the synchronization mechanism is only used on the client side than synchronization is not enforced. However, with client-side synchronization the client can compose different operations on the shared resource into a larger synchronized operation.
- **Enforced Isolation**. If the synchronization mechanism only enforces synchronized access to the root object then any leaked reference to a nested object will not be synchronized and can potentially lead to race conditions.

Table 2 summarises our findings and classifies these three patterns according to their properties. Note that we do not consider isolated domains in this table because isolated domains can only be directly accessed by a single actor and are thus not suited to represent shared state. A technical report with the full validation can be found online [29]. In addition to what is presented in this paper, a library for Scala was created for adding support for shared domains. For each pattern we identified a representative example and translated this example to a version using our shared domain library in Scala.

| | **Occurrences** | **No CPS** | **Deadlock free** | **Parallel reads** | **Enforced Synchronization** | **Composable Interface** | **Enforced Isolation** |
|---|---|---|---|---|---|---|---|
| Server-side Lock | 166 | ✓ | × | ✓ | ✓ | × | × |
| Client-side Lock | 38 | ✓ | × | ✓ | × | ✓ | × |
| Delegate Actor | 28 | × | ✓ | × | ✓ | × | × |
| Immutable Domains | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Observable Domains | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Shared Domains | - | ×/✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 2: The different synchronization patterns and their properties

The advantage of using blocking synchronization mechanisms such as locks is that client-side processes have direct synchronous access to the shared state. This means that there is **no need to apply a CPS** transformation on the client-side code that is accessing the shared resource. Using a non-blocking mechanism such as a delegate actor forces the client to apply CPS. Immutable and observable domains allow direct synchronously access to their object heap, avoiding the need for CPS. However, observable domains can only be mutated by their owners, this means that any composite operation involving write operations will still have to be CPS transformed. Shared domains require one level of CPS because acquiring a view is an asynchronous operation. However, during a single view an actor has unlimited synchronous access to the shared resource, avoiding any extra levels of CPS. The advantage of using a non-blocking synchronization mechanism is that **deadlocks are avoided**. As the domain model does not allow for any blocking operations

it is completely deadlock free. The advantage of using a server-side synchronization mechanism such as a server-side lock or a delegate actor is that **synchronization is enforced** upon each access of the shared resource. The downside of this compared to client-side synchronization mechanisms is that clients cannot **compose different operations on the shared resource** in a larger synchronous operation. Domains combine these advantages as synchronization is enforced (i. e. for shared domains, accessing the domain outside of a view results in a runtime error) and during a single turn the isolated turn principle guarantees that multiple operations on the shared resource can be combined in a larger synchronous operation. As with Scala, locks are usually only associated with a single object. That means that **isolation of the whole object graph of a shared resource** cannot be guaranteed. While using a delegate actor would usually guarantee isolation, because the Scala actor library is *impure* this is not guaranteed. With domains isolation is enforced, any synchronized access applies to the whole object graph in the domain.

## 6. Related work

The engineering benefits of semantically coarse-grained synchronization mechanisms in general, and the restrictions of the actor model have been recognized by others. In particular the notions of *domain*-like and *view*-like constructs have been proposed before. We divide the existing related work 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 domains*

*Ribbons.* Another approach similar to our notion of domains is the notion of ribbons by Hoffman et al. [30] 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. Different threads are grouped into ribbons and access rights are defined on those ribbons. 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. As a consequence, they do not provide the same guarantees for freedom of low-level data races and deadlocks. 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 [31] the programmer has to use effect annotations to indicate which 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.

*PGAS languages.* In partitioned global address space (PGAS) languages such as X10 [32], Chapel [33] and Fortress [34] the global memory is partitioned and each partition is local to a specific process or thread. For example, in X10, a place corresponds to a single isolated address space. By default, an X10 process can only access a location in a local place. However, using the `at(place){...}` construct allows a process to *send* code across places while maintaining the mapping between the global address space and each local address space. X10 is mainly used in a distributed setting, however, it is possible to run with multiple places installed in a single machine. The main difference between places and domains is that the code executed in a domain is not *sent* to that domain but rather executed locally by the actor that is accessing the domain. This has the benefit that the actor has access to both the remote domain as well as its own local isolated domain.

*ETS Tables.* Erlang's [1] sequential subset is a purely functional language. However, Erlang does have some support for shared mutable state in the form of *Erlang Term Storage Tables* which allows storing tuples of key-value pairs. Each table is created by a process (the owner of the table) and when the process terminates, the table is automatically destroyed. Upon creating a table, the process can set its access rights to either public, protected or private. Public is similar to shared domains as any process can read and write to the table. Protected is similar to observable domains, only the owner of the table can read and write to the table. Other processes can only read from the table. Private is similar to isolated domains as only the owner can read and write to the table. Different ETS tables are isolated from one another and while the access rights are very similar to the different domains, they are not guaranteed on turn boundaries. Any write to a public or protected table will be immediately visible by other processes which violates the isolated turn principle.

*Related work on view-like abstractions*

*Demsky views.* Closely related to our model are the views of Demsky et al. [35], 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 in refining 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 [36]. The Axum project shares 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 [8] is middleware for Java that provides an actor abstraction on top of threads. It provides the notion of *Coordination objects* to avoid data races similar to views. However, the overall reasoning about thread safety is hampered since the use of coordination objects is not enforced. Furthermore, coordination objects are proxy objects that serialize 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.

*Other related work*

*Semantics-preserving Sharing Actors.* Lesani et al. [37] realize a sharing actor theory that is very similar to observable domains. Each actor is able to share a number of abstract data types with other actors by keeping

multiple versions by storing the update function applied. Interestingly, they also chose turn boundaries as the boundaries for isolation. Unfortunately, they only apply their theory to a small number of abstract data types and containers. The domain model is an attempt at a more generic abstraction.

*Zero-copy message passing.* There is a body of related work that care about sharing state through efficiently passing the arguments of a message between actors by reference [15, 38, 39]. This class of research wants to avoid the cost of deep copying a data-structure when it is passed by reference. While this is useful in the context of ownership transfer, it does not really solve the state-sharing issue in the actor model. While objects can migrate between different actors, there is always only one owner for each object. It is impossible for different actors to read from a shared data structure in parallel.

*Parallel Actor Monitors.* The strong restrictions of the actor model with regard to shared state and parallelism have also been discussed earlier. One example is Parallel Actor Monitors [24] (PAM). PAM enables parallelism inside a single actor by evaluating different messages that are tagged as *read-only* 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 to 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.

## 7. Conclusion

In this paper we have shown that pure actor languages provide strong safety guarantees but, because of the strong isolation between different actors, lack the necessary language abstractions to model shared resources in an expressive way. However, we also show that strict isolation is not a necessary prerequisite to provide these guarantees, as demonstrated by our domain abstractions. Our abstractions tackle the problem of expressing access to shared resources on two levels:

**Safety** With the introduction of the domain model actors are no longer strictly isolated software entities. However, the domains themselves are still completely isolated from one another. We have shown that the domain model maintains the strong language-enforced guarantees of the pure actor model and prevents low-level data races and deadlocks by design. We have also shown that the domain model maintains the isolated turn property which is important for formal reasoning about program semantics, and provides additional guarantees to facilitate application development.

**Expressiveness** When designing software, many of the concepts can be divided in either passive or active software entities. The strict isolation of pure actor languages forces developers to model a passive shared resource by encapsulating it in a separate shared actor, which is an active software entity. The domain model allows the programmer to make better distinction between the two by representing shared passive software entities as domain objects and active software entities as actors. On top of that developers can choose between four types of domains for specifying the access capabilities of the different software entities using that shared resource. Immutable domains can be used for representing read-only shared resources that can be freely shared between the different actors. Isolated domains can be used when full isolation of the resource is required, only a single actor will be able to read and write to objects in an isolated domain. Observable domains can be used to represent a resource that is "owned" by a single actor but can be exposed as a read-only resource to other actors. Shared domains can be used in the case where any actor is free to read from and write to the shared resource.

We show that by unifying domains and object heaps, we can neatly integrate the domain model within the communicating event-loop actor model without compromising on its safety guarantees. This paper gives a taxonomy that led to the design of the four types of domains. The common ground for each of the domains is that they provide coordinated synchronous access to a heap of shared objects. Each of the domain types was implemented in a communicating event-loop language called SHACL. We discussed

syntax, operational semantics and use-cases of each domain type. Additionally, we have shown that the synchronization mechanisms used in existing projects that use actors can be translated to domains with the added benefit that the domain model can combine server-side and client-side synchronization in a safe and expressive way.


[1] J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent Programming in ERLANG (2nd ed.), Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[2] C. Varela, G. Agha, Programming Dynamically Reconfigurable Open Systems with SALSA, SIGPLAN Notices 36 (12) (2001) 20–34.

[3] M. S. Miller, E. D. Tribble, J. Shapiro, Concurrency Among Strangers: Programming in E As Plan Coordination, in: TGC'05: Proceedings of the 1st International Conference on Trustworthy Global Computing, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 195–229.

[4] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter, AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad Hoc Networks, in: SCCC'07: Proceedings of the 26th International Conference of the Chilean Society of Computer Science, IEEE Computer Society, Washington, DC, USA, 2007, pp. 3–12.

[5] S. Srinivasan, A. Mycroft, Kilim: Isolation-Typed Actors for Java, in: ECOOP'08: Proceedings of the 22nd European Conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 104–128.

[6] M. Astley, The actor foundry: A java-based actor programming environment (1998-99).
URL http://osl.cs.uiuc.edu/foundry

[7] M.-W. Jang, The actor architecture manual (March 2004).
URL http://osl.cs.uiuc.edu/aa

[8] 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.
URL http://www-sop.inria.fr/oasis/proactive/doc/ProgrammingComposingDeploying.pdf

[9] C. Plotnikov, Asyncobjects framework., http://asyncobjects.sourceforge.net/ (2007).

[10] S. R. J.-P. Arcangeli, F. Migeon, Javact : a java middleware for mobile adaptive agents (February 2008).
URL http://www.irit.fr/PERSONNEL/SMAC/arcangeli/JavAct.html

[11] M. Rettig, Jetlang (2008-09).
URL http://code.google.com/p/jetlang/

[12] W. Zwicky, Aj: A systems for buildings actors with java (2008).

[13] P. Haller, M. Odersky, Actors That Unify Threads and Events, in: COORDINATION'07: Proceedings of the 9th International Conference on Coordination Models and Languages, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 171–190.

[14] J. Allen, Effective Akka, O'Reilly Media, Inc., 2013.

[15] P. Haller, M. Odersky, Capabilities for Uniqueness and Borrowing, in: ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 354–378.

[16] S. Tasharofi, P. Dinges, R. E. Johnson, Why do scala developers mix the actor model with other concurrency models?, in: ECOOP'13: Proceedings of the 27th European conference on Object-Oriented Programming, Springer-Verlag, 2013.

[17] J. De Koster, S. Marr, T. D'Hondt, T. Van Cutsem, Tanks: multiple reader, single writer actors, in: AGERE! 2013: Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control, ACM Request Permissions, New York, New York, USA, 2013, pp. 61–68.

[18] J. D. Koster, S. Marr, T. D'Hondt, T. V. Cutsem, Domains: Safe sharing among actors, Science of Computer Programming 98, Part 2 (0) (2015) 140–158, special Issue on Programming Based on Actors, Agents and Decentralized Control. doi:10.1016/j.scico.2014.02.008.

[19] J. De Koster, The shacl programming language, http://soft.vub.ac.be/~{}jdekoste/shacl/ (2014).

[20] 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.

[21] R. E. Johnson, B. Foote, Designing reusable classes, Journal of Object-Oriented Programming 1 (2) (1988) 22–35.

[22] Microsoft, C# Reference: await (2013).
URL https://msdn.microsoft.com/en-us/library/hh156528.aspx

[23] P. Haller, J. Zaugg, SIP-22 – Async (June 2013).
URL http://docs.scala-lang.org/sips/pending/async.html

[24] C. Scholliers, E. Tanter, W. De Meuter, Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model, Sci. Comput. Program. 80 (2014) 52–64. doi:10.1016/j.scico.2013.03.011.
URL http://dx.doi.org/10.1016/j.scico.2013.03.011

[25] T. Van Cutsem, E. Gonzalez Boix, C. Scholliers, A. Lombide Carreton, D. Harnie, K. Pinte, W. De Meuter, AmbientTalk: programming responsive mobile peer-to-peer applications with actors, Computer Languages, Systems & Structures 40 (34) (2014) 112–136. doi:http://dx.doi.org/10.1016/j.cl.2014.05.002.

[26] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing Active Objects to Concurrent Components, in: ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 275–299.

[27] N. Shavit, D. Touitou, Software Transactional Memory, in: PODC'95: Proceedings of the 14th Symposium on Principles of Distributed Computing, ACM, New York, NY, USA, 1995, pp. 204–213.

[28] D. Perelman, R. Fan, I. Keidar, On Maintaining Multiple Versions in STM, in: PODC'10: Proceedings of the 29th Symposium on Principles of Distributed Computing, ACM, New York, NY, USA, 2010, pp. 16–25.

[29] J. De Koster, The Domain Model: Operational Semantics, Tech. rep. (2015).

URL http://soft.vub.ac.be/Publications/2015/vub-soft-tr-15-07.pdf
[30] K. J. Hoffman, H. Metzger, P. Eugster, Ribbons: A Partially Shared Memory Programming Model, SIGPLAN Notices 46 (10) (2011) 289–306.
[31] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. Vakilian, A Type and Effect System for Deterministic Parallel Java, SIGPLAN Notices 44 (10) (2009) 97–116.
[32] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An Object-oriented Approach to Non-uniform Cluster Computing, SIGPLAN Notices 40 (10) (2005) 519–538.
[33] B. Chamberlain, D. Callahan, H. Zima, Parallel programmability and the chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312. doi:10.1177/1094342007078442.
URL http://dx.doi.org/10.1177/1094342007078442
[34] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, Jr, S. Tobin-Hochstadt, The Fortress Language Specification, version 1.0, Tech. rep., Sun Microsystems, Inc. (2008).
[35] B. Demsky, P. Lam, Views: Object-inspired Concurrency Control, in: ICSE'10: Proceedings of the 32nd International Conference on Software Engineering, ACM, New York, NY, USA, 2010, pp. 395–404.
[36] Microsoft, Axum programming language (2008-09).
URL http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx
[37] M. Lesani, A. Lain, Semantics-preserving sharing actors, in: AGERE! 2013: Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control, ACM Request Permissions, New York, New York, USA, 2013, pp. 69–80.
[38] S. Negara, R. K. Karmani, G. Agha, Inferring ownership transfer for efficient message passing, in: PPoPP'11: Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming, ACM, 2011, pp. 81–90.
[39] O. Gruber, F. Boyer, Ownership-based isolation for concurrent actors on multi-core machines, in: Proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP'13, 2013, pp. 281–301.
[40] M. Felleisen, R. Hieb, The Revised Report on the Syntactic Theories of Sequential Control and State, Theoretical Computer Science 103 (2) (1992) 235–271.

## Appendix A. An Operational Semantics for a Significant Subset of Shacl

The exposition of the domain model in Section 4 was largely informal. In this appendix we provide a small step operational semantics for a small but significant subset of SHACL, named SHACL-LITE. The aim of this operational semantics is to serve as a reference specification of the semantics of our language abstractions regarding domains. The operational semantics of SHACL-LITE was primarily based on an operational semantics for the AmbientTalk language [25] which in turn was based on that of the Cobox [26] model. Our operational semantics starts off by modeling actors, objects and event-loops for a small communicating event-loop language. On top of that we build semantic rules for adding the four types of domains.

### *Appendix A.1. Introduction*

The full operational semantics is built in four steps. In Appendix A.2 we build an operational semantics for a regular communicating event-loop language. Isolated domains are interchangeable with the object heap of a traditional event-loop actor as they have equivalent properties. Thus, in the first version, the object heaps of actors are already replaced with the associated isolated domains. In the subsections that follow we extend the operational semantics with immutable, observable and shared domains. Each extension of the semantics can be done with minimal changes to the original rules. If a semantic rule is replaced this will be announced in the text.

### *Appendix A.2. Basic SHACL-LITE, Actors and Their Isolated Domains*

In this subsection we start off by modeling a small event-loop actor model. This first version models objects, actors, event-loops, and object heaps. The object heap of an actor is modeled as an isolated domains. The addition of isolated domains is not visible in the syntax as no syntax was added for creating new isolated domain. However, the fact that object heaps are already separated from actors allows us to extend these semantics with additional domains with minimal changes to the existing rules. It also allows for a uniform definition of object ownership.

**Semantic Entities of Shacl**

| | | | |
|---|---|---|---|
| $K \subseteq$ **Configuration** | ::= | $\mathcal{K}\langle A, D\rangle$ | Configurations |
| $a \in A \subseteq$ **Actor** | ::= | $\mathcal{A}\langle \iota_a, Q, e\rangle$ | Actors |
| $D \subseteq$ **Domain** | ::= | $I$ | Domains |
| $I \subseteq$ **Isolated** | ::= | $\mathcal{I}\langle \iota_a, O\rangle$ | Isolated Domains |
| $o \in O \subseteq$ **Object** | ::= | $\mathcal{O}\langle \iota_o, F, M\rangle$ | Objects |
| $m \in$ **Message** | ::= | $\mathcal{M}\langle r, m, \overline{v}\rangle$ | Messages |
| $Q \subseteq$ **Queue** | ::= | $\overline{m}$ | Queues |
| $M \subseteq$ **Method** | ::= | $\mathrm{m}(\overline{x})\{e\}$ | Methods |
| $F \subseteq$ **Field** | ::= | $f := v$ | Fields |
| $v \in$ **Value** | ::= | $r \mid \mathrm{null}$ | Values |
| $r \in$ **Reference** | ::= | $\iota_d.\iota_o$ | References |

$\iota_o \in$ **ObjectId**, $\iota_d \in$ **DomainId**, $\iota_a \in$ **IsolatedId**, **IsolatedId** $\subseteq$ **DomainId**

Figure A.7: Semantic entities of SHACL-LITE

*Appendix A.2.1. Semantic Entities*

Figure A.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 $\iota_a$, $\iota_d$ and $\iota_o$ respectively.

In SHACL-LITE a **Configuration** consists of a set of live actors, $A$ and a set of domains, $D$. A single configuration represents the whole state of a SHACL-LITE program in a single step. In SHACL-LITE each **Actor** has an identity $\iota_a$. Currently the only type of domain that is represented is the **Isolated** domain, $I$. All actors are associated with a single isolated domain with the same identity as the actor. This isolated 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 isolated Ids is a subset of the set of domain Ids. Each actor also has a queue of pending messages $Q$, and the expression $e$ it is currently evaluating, i. e., reducing. An **Object** has an identity $\iota_o$, 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. The **Queue** used by the event-loop of an actor is an ordered list of pending messages. 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 $r$ or `null`. A reference identifies an object located within a certain domain. In this version of the semantics only isolated domains exist.

**Shacl Syntax**

**Syntax**

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

**Runtime Syntax**

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

**Evaluation Contexts**

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

**Syntactic Sugar**

$$
\begin{array}{lcll}
e\ ;\ e' & \stackrel{\text{def}}{=} & \text{let } x = e \text{ in } e' & x \notin \text{FV}(e') \\
\lambda\overline{x}.e & \stackrel{\text{def}}{=} & \text{let } x_{this} = \text{this in} & x_{this} \notin \text{FV}(e) \\
 & & \quad \text{object } \{ & \\
 & & \quad\quad \text{apply}(\overline{x})\{[x_{this}/\text{this}]e\} & \\
 & & \quad \} & \\
e(\overline{e}) & \stackrel{\text{def}}{=} & e.\text{apply}(\overline{e}) &
\end{array}
$$

Figure A.8: Syntax of SHACL-LITE

*Appendix A.2.2.* 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 ($e.m(\overline{e})$) and asynchronously ($e \leftarrow m(\overline{e})$). In the context of a method, the pseudo variable `this` refers to the enclosing object. `this` cannot be used as a parameter name in methods or redefined using `let`. New actors can be spawned using the `actor` literal expression. This creates a fresh actor that is linked with a fresh isolated domain by sharing the same identifier. This isolated domain is instantiated with a single new object, with the given fields and methods, in its heap. The newly created actor executes in parallel with the other actors in the system. Expressions contained in actor literals may not refer to lexically enclosing variables, apart from the `this`-pseudo variable. That is, all variables have to be bound except `this`, which means $FV(e) \subseteq \{$ `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.

*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, $r$, and object literals that are annotated with the domain identifier of their lexically enclosing domain. This annotation is required so that upon object creation each object gets associated with the appropriate domain.

*Evaluation contexts.* We use evaluation contexts [40] 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.

*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.

**Substitution Rules**

$$
\begin{array}{rcl@{\qquad}rcl}
[v/x]x' & = & x' & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{e\} \text{ if } x \in \overline{x} \\
[v/x]x & = & v & [v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{[v/x]e\} \text{ if } x \notin \overline{x} \\
[v/x]e.f & = & ([v/x]e).f & [v/x]e.f := e & = & ([v/x]e).f := [v/x]e \\
[v/x]\text{null} & = & \text{null} & [v/x]e \leftarrow m(\overline{e}) & = & [v/x]e \leftarrow m([v/x]\overline{e}) \\
[v/x]e.m(\overline{e}) & = & [v/x]e.m([v/x]\overline{e}) & & & \\
\end{array}
$$

$$
\begin{array}{rcl}
[v/x]\text{let } x' = e \text{ in } e & = & \text{let } x' = [v/x]e \text{ in } [v/x]e \\
[v/x]\text{let } x = e \text{ in } e & = & \text{let } x = [v/x]e \text{ in } e \\
[v/x]\text{actor}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{actor}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{immutable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{immutable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{observable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{observable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{shared}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{shared}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{object}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := [v/x]e}, \overline{[v/x]m(\overline{x})\{e\}}\} \text{ if } x \neq \text{this} \\
[v/\text{this}]\text{object}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}
\end{array}
$$

Figure A.9: Substitution rules: $x$ denotes a variable name or the pseudovariable this.

*Appendix A.2.3. Substitution and Tagging Rules*

*Substitution rules.*

Figure A.9 lists the different rules for propagating variable/value substitutions. For completeness, the substitution rules for the different domains have already been included at this stage. In most cases the variable is substituted by the value within the different subexpressions of the compound expression. Expressions contained in the actor literal and the different domain literals have to be lexically closed, this means that subexpressions are not substituted.

**Tagging Rules**

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]_{\iota_d} & = & x & [\![m(\overline{x})\{e\}]\!]_{\iota_d} & = & m(\overline{x})\{[\![e]\!]_{\iota_d}\} \\
[\![e.f]\!]_{\iota_d} & = & ([\![e]\!]_{\iota_d}).f & [\![e.f := e]\!]_{\iota_d} & = & ([\![e]\!]_{\iota_d}).f := [\![e]\!]_{\iota_d} \\
[\![\mathrm{null}]\!]_{\iota_d} & = & \mathrm{null} & [\![e \leftarrow m(\overline{e})]\!]_{\iota_d} & = & [\![e]\!]_{\iota_d} \leftarrow m([\![\overline{e}]\!]_{\iota_d}) \\
[\![e.m(\overline{e})]\!]_{\iota_d} & = & [\![e]\!]_{\iota_d}.m([\![\overline{e}]\!]_{\iota_d}) & & &
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathrm{let}\ x = e\ \mathrm{in}\ e]\!]_{\iota_d} & = & \mathrm{let}\ x = [\![e]\!]_{\iota_d}\ \mathrm{in}\ [\![e]\!]_{\iota_d} \\
[\![\mathrm{object}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \mathrm{object}_{\iota_d}\{\overline{f := [\![e]\!]_{\iota_d}}, \overline{m(\overline{x})\{[\![e]\!]_{\iota_d}\}}\} \\
[\![\mathrm{immutable}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \mathrm{immutable}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\} \\
[\![\mathrm{observable}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \mathrm{observable}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\} \\
[\![\mathrm{shared}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \mathrm{shared}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\} \\
[\![\mathrm{actor}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \mathrm{actor}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e\}}\}
\end{array}
$$

Figure A.10: Runtime object tagging rules.

*Tagging rules.*

Every object literal is tagged at runtime with the identifier of its lexically enclosing domain, $\iota_d$, using the $\mathrm{object}_{\iota_d}$ runtime syntax. Figure A.10 lists a number of rules on how this tag is propagated through the different subexpressions. Any compound expression simply propagates the substitution to its subexpressions except for the actor literal and the different domain literals.

*Appendix A.2.4. Reduction Rules*

*Notation.* Actor heaps $O$ are sets of objects. To lookup and extract values from a set $O$, we use the notation $O = O' \uplus \{o\}$. This splits the set $O$ into a singleton set containing the desired object $o$ and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \mathrm{m}$ deconstructs a sequence $Q$ into a subsequence $Q'$ and the last element m. In SHACL-LITE, queues are sequences of messages and are processed right-to-left, meaning that the last message 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:

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

The initial configuration contains a *main actor* and its associated empty isolated domain. Every lexically nested object expression in the program is annotated with the domain identifier of the main isolated domain using the $[\![e]\!]_{\iota_a}$ syntax.

$$\text{(LET)}\quad \mathcal{A}\langle \iota_a, Q, e_\Box[\text{let } x = v \text{ in } e]\rangle \to_a \mathcal{A}\langle \iota_a, Q, e_\Box[[v/x]e]\rangle$$

$$\text{(PROCESS-MESSAGE)}\quad \mathcal{A}\langle \iota_a, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \to_a \mathcal{A}\langle \iota_a, Q, \iota_a.\iota_o.m(\overline{v})\rangle$$

$$\text{(INVOKE)}\quad \frac{r = \iota_a.\iota_o \qquad \mathcal{I}\langle \iota_a, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \text{m}(\overline{x})\{e\} \in M}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[r.m(\overline{v})]\rangle\}, D\rangle \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\}, D\rangle}$$

$$\text{(FIELD-ACCESS)}\quad \frac{\mathcal{I}\langle \iota_a, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f := v \in F}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_a.\iota_o.f]\rangle\}, D\rangle \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[v]\rangle\}, D\rangle}$$

$$\text{(FIELD-UPDATE)}\quad \frac{o = \mathcal{O}\langle \iota_o, F \uplus \{f := v'\}, M\rangle \quad o' = \mathcal{O}\langle \iota_o, F \cup \{f := v\}, M\rangle \quad D = D' \uplus \{\mathcal{I}\langle \iota_a, O \uplus \{o\}\rangle\} \quad D'' = D' \cup \{\mathcal{I}\langle \iota_a, O \cup \{o'\}\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_d.\iota_o.f := v]\rangle\}, D\rangle \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[v]\rangle\}, D''\rangle}$$

$$\text{(CONGRUENCE)}\quad \frac{a \to_a a'}{\mathcal{K}\langle A \uplus \{a\}, D\rangle \to_k \mathcal{K}\langle A \cup \{a'\}, D\rangle}$$

Figure A.11: Actor-local reduction rules and congruence.

*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 evaluating (reducing) 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. A value cannot be further reduced and the actor sits idle until it receives a new message.

We now summarize the actor-local reduction rules in Figure A.11:

- LET: Reducing a "let"-expression simply substitutes the value of $x$ for $v$ in $e$.
- PROCESS-MESSAGE: this rule describes the processing of incoming asynchronous messages 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 its associated isolated domain (with the same domain identifier, $\iota_a$).
- FIELD-ACCESS, FIELD-UPDATE. It is *only* possible for an actor to access or update a field of an object within its associated isolated domain. 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.

- CONGRUENCE: this rule simply connects the actor local reduction rules to the global configuration reduction rules.

(NEW-OBJECT)

$$\frac{\begin{array}{c}\iota_o \text{ fresh} \qquad r = \iota_a.\iota_o \\ o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\rangle\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{object}_{\iota_a}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D \uplus \{\mathcal{I}\langle \iota_a, O\rangle\}\rangle \\ \quad \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D \cup \{\mathcal{I}\langle \iota_a, O \cup \{o\}\rangle\}\rangle\end{array}}$$

(NEW-ACTOR)

$$\frac{\begin{array}{cc}\iota_{a'}, \iota_o \text{ fresh} & o = \mathcal{O}\langle \iota_o, \overline{f := \mathrm{null}}, \overline{\mathrm{m}(\overline{x})\{[\![e']\!]_{\iota_{a'}}\}}\rangle \\ r = \iota_{a'}.\iota_o & a = \mathcal{A}\langle \iota_{a'}, \emptyset, \overline{r.f := [r/\mathrm{this}][\![e]\!]_{\iota_{a'}}}\rangle\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{actor}\{\overline{f := e}, \overline{\mathrm{m}(\overline{x})\{e'\}}\}]\rangle\}, D\rangle \\ \quad \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle, a\}, D \cup \{\mathcal{I}\langle \iota_{a'}, \{o\}\rangle\}\rangle\end{array}}$$

Figure A.12: Creational rules

*Rules for object and actor literals.* We summarize the creation reduction rules in Figure A.12:

- NEW-OBJECT: An object expression can only be reduced once its field initialization expressions have been reduced to a value. All object expressions are tagged with the domain id of the lexically enclosing domain. The effect of reducing an object literal expression is the addition of a new object to the heap of that domain. The literal expression reduces to a domain reference $r$ to the new object.
- NEW-ACTOR: when an actor $\iota_a$ reduces an actor literal expression, a new actor $\iota_{a'}$ is added to the set of actors of the configuration. A newly created isolated domain is associated with that actor. The new domain's heap consists of a single new object $\iota_o$ whose fields and methods are described by the literal expression. The $[\![e]\!]_{\iota_d}$ syntax makes sure that all lexically nested object expressions are tagged with the domain id of the newly created domain. The actor literal expression reduces to a domain reference to the new object, allowing the actor that created the new actor to communicate further with that actor.

(LOCAL-ASYNCHRONOUS-SEND)

$$\begin{array}{l}\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle \\ \to_a \mathcal{A}\langle \iota_a, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\Box[\mathrm{null}]\rangle\end{array}$$

(REMOTE-ASYNCHRONOUS-SEND)

$$\frac{\begin{array}{c}A = A' \uplus \{\mathcal{A}\langle \iota_{a'}, Q', e'\rangle\} \\ A'' = A' \cup \{\mathcal{A}\langle \iota_{a'}, \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}\rangle \cdot Q', e'\rangle\}\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle\}, D\rangle \\ \quad \to_k \mathcal{K}\langle A'' \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{null}]\rangle\}, D\rangle\end{array}}$$

Figure A.13: Asynchronous message rules

*Asynchronous communication reductions.* We summarize the asynchronous communication reduction rules in Figure A.13:

- LOCAL-ASYNCHRONOUS-SEND: an asynchronous message sent to a *local* object (i. e., an object owned by the isolated domain of 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`.

- REMOTE-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a remote isolated reference, i. e., an isolated 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`.

*Appendix A.3. Immutable Domains*

An immutable domain is an object heap of immutable objects. Each lexically nested object expression will reduce to an object that belongs to that immutable domain. This is achieved by the tagging rules as described in Appendix A.2.3. Immutability of objects owned by an immutable domain is achieved by *not* specifying a reduction rule for updating the field of an object owned by an immutable domain. The addition of immutable domains does not alter any of the existing reduction rules.

**Semantic Entities of Shacl**

$$
\begin{array}{rcll}
D \subseteq \textbf{Domain} & ::= & C \cup I & \text{Domains} \\
C \subseteq \textbf{Immutable} & ::= & \mathcal{C}\langle \iota_c, O \rangle & \text{Immutable Domains} \\
\iota_c \in \textbf{ImmutableId}, \textbf{ImmutableId} \cup \textbf{IsolatedId} \subseteq \textbf{DomainId} & & &
\end{array}
$$

Figure A.14: Additional semantic entities for immutable domains

*Appendix A.3.1. Semantic Entities*

The set of domains is extended with the set of immutable domains. An **Immutable** domain has an identifier, $\iota_c$ and an object heap. **ImmutableId** and **IsolatedId** are a distinct subset of **DomainId**.

**Shacl Syntax**

**Syntax**

$$e \in E \subseteq \textbf{Expression} \quad ::= \quad \ldots \mid \text{immutable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}$$

**Evaluation Contexts**

$$e_\square \quad ::= \quad \ldots \mid \text{immutable}\{\overline{f := v}, f := e_\square, \overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}$$

Figure A.15: Additional syntax for immutable domains

*Appendix A.3.2. Syntax*

The SHACL-LITE syntax is extended with syntax to create new immutable domains. Similar to the object syntax, the field initialiser expressions of an isolated domain expression need to be reduced to a value from left to right before the isolated domain literal can be further reduced. An additional evaluation context was added to specify this behavior.

(IMMUTABLE-INVOKE)

$$\frac{r = \iota_c.\iota_o \qquad \mathcal{C}\langle \iota_c, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \mathrm{m}(\overline{x})\{e\} \in M}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[r.m(\overline{v})]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[[r/\mathrm{this}][\overline{v}/\overline{x}]e]\rangle\}, D\rangle\end{array}}$$

(IMMUTABLE-FIELD-ACCESS)

$$\frac{\mathcal{C}\langle \iota_c, O\rangle \in D \qquad \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f := v \in F}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_c.\iota_o.f]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[v]\rangle\}, D\rangle\end{array}}$$

Figure A.16: Immutable Domain Actor-Local Reduction Rules.

*Appendix A.3.3. Reduction Rules*

*Actor-local reductions.* We summarize the actor-local reduction rules in Figure A.16:

- IMMUTABLE-INVOKE: Similar to a method invocation on an isolated domain reference, the method $m$ is simply looked up 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`. However, in this case any actor can invoke a method on an immutable domain reference, regardless of the identifier of the domain.
- IMMUTABLE-FIELD-ACCESS. Any actor can access a field of an immutable domain object. The object is looked up in the appropriate immutable domain and the field access is reduced to the associated value.
- IMMUTABLE-FIELD-UPDATE. There is no rule specified for field updates on immutable domain references. A field update expression on an immutable domain reference will not be further reduced and lead to a stuck state.

(NEW-IMMUTABLE-OBJECT)

$$\frac{\iota_o \text{ fresh} \qquad r = \iota_c.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\rangle}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{object}_{\iota_c}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D \uplus \{\mathcal{C}\langle \iota_c, O\rangle\}\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D \cup \{\mathcal{C}\langle \iota_c, O \cup \{o\}\rangle\}\rangle\end{array}}$$

(NEW-IMMUTABLE-DOMAIN)

$$\frac{\iota_c, \iota_o \text{ fresh} \qquad r = \iota_c.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{[\![e]\!]_{\iota_c}\}}\rangle}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{immutable}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D \cup \{\mathcal{C}\langle \iota_c, \{o\}\rangle\}\rangle\end{array}}$$

Figure A.17: Immutable Creational Rules

*Rules for object and immutable domain literals.* We summarize the immutable domain creation reduction rules in Figure A.17:

- NEW-IMMUTABLE-OBJECT: An object expression can only be reduced once its field initialization expressions have been reduced to a value. The effect of reducing an object literal expression is the addition of a new object to the heap of the immutable domain. The literal expression reduces to a domain reference $r$ to the new object.

- NEW-IMMUTABLE-DOMAIN: A domain literal will reduce to the construction of a new immutable domain with a single object in its heap. That domain is added to the set of domains in the configuration. Similarly to the rule for NEW-OBJECT, the immutable domain expression can only be further reduced once its field initialization expressions have been reduced to a value. The domain expression reduces to an immutable domain reference $r$ to the newly created object. $[\![e]\!]_{\iota_c}$ denotes a transformation that makes sure that all lexically nested object expressions are annotated with the domain id of the newly created domain.

(IMMUTABLE-ASYNCHRONOUS-SEND)
$$\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_c.\iota_o \leftarrow m(\overline{v})]\rangle$$
$$\rightarrow_a \mathcal{A}\langle \iota_a, \mathcal{M}\langle \iota_c.\iota_o, m, \overline{v}\rangle \cdot Q, e_\Box[\text{null}]\rangle$$

Figure A.18: Immutable Asynchronous Message Reduction Rules

*Asynchronous communication reductions.* We summarize the asynchronous communication reduction rule in Figure A.18:

- IMMUTABLE-ASYNCHRONOUS-SEND: an asynchronous message sent to an immutable object simply appends a new message to the end of the sender's own message queue. The asynchronous message send itself immediately reduces to `null`.

*Appendix A.4. Observable Domains*

An observable domain object is synchronously readable by every actor as long as they have obtained a reference to that object. To ensure that the isolated turn principle remains valid, any actor always reads values from a consistent snapshot of the object heap of an observable domain. That snapshot is stored in the semantic function, $f$. The addition of observable domains alters the reduction rule for processing messages, PROCESS-MESSAGE, such that each actor takes a snapshot of the various observable domains at the start of a turn and commits any changes made at the end of a turn.

**Semantic Entities of Shacl**

| | | | |
|---|---|---|---|
| $D \subseteq$ **Domain** | ::= | $C \cup I \cup B$ | Domains |
| $B \subseteq$ **Observable** | ::= | $\mathcal{B}\langle \iota_b, \iota_a, f, O\rangle$ | Observable Domains |
| $\iota_b \in$ **ObservableId**, **IsolatedId**$\cup$**ImmutableId**$\cup$**ObservableId** $\subseteq$ **DomainId** | | | |

Figure A.19: Additional semantic entities for observable domains

*Appendix A.4.1. Semantic Entities*

The set of domains is extended with the set of observable domains. An **Observable** domain has an identifier ($\iota_b$) and an identifier that specifies the owner of the domain ($\iota_a$). It also has a semantic funtion, $f$, that maps actor identifiers to a temporary snapshot of the object heap of the observable domain. Each time an actor accesses an observable domain, that actor will read from that snapshot. Lastly, it has an object heap, $O$, that represents the latest consistent version of the objects in the domain. **ImmutableId**, **IsolatedId** and **ObservableId** are distinct subsets of **DomainId**.

**Shacl Syntax**

**Syntax**

$e \in E \subseteq$ **Expression** ::= $\ldots \mid \text{observable}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}$

**Runtime Syntax**

$e$ ::= $\ldots \mid \text{commit}$

**Evaluation Contexts**

$e_\square$ ::= $\ldots \mid \text{observable}\{\overline{f := v}, f := e_\square, \overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}$

Figure A.20: Additional syntax for observable domains

*Appendix A.4.2. Syntax*

The SHACL-LITE syntax is extended with a new syntax expression to create new observable domains. Additional runtime syntax was added to ensure that an actor commits any changes made to its observable domains at the end of each turn. Similar to the object syntax, the field initialiser expressions of an observable domain expression need to be reduced to a value from left to right before the observable domain literal can be further reduced. An additional evaluation context was added to specify this behavior.

(PROCESS-MESSAGE)
$$\frac{D' = \mathit{snapshot}(\iota_a, D)}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, \iota_a.\iota_o.m(\overline{v}); \text{commit}\rangle\}, D'\rangle\end{array}}$$

(COMMIT)
$$\frac{D' = \mathit{commit}(\iota_a, D)}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, \text{commit}\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, \text{null}\rangle\}, D'\rangle\end{array}}$$

(OBSERVABLE-INVOKE)
$$\frac{\begin{array}{c}r = \iota_b.\iota_o \qquad \mathcal{B}\langle \iota_b, \iota_{a'}, f, O'\rangle \in D \qquad f(\iota_a) = O \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \text{m}(\overline{x})\{e\} \in M\end{array}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\square[r.m(\overline{v})]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\}, D\rangle\end{array}}$$

(OBSERVABLE-FIELD-ACCESS)
$$\frac{\begin{array}{c}\mathcal{B}\langle \iota_b, \iota_{a'}, f, O'\rangle \in D \qquad f(\iota_a) = O \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f := v \in F\end{array}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\square[\iota_b.\iota_o.f]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[v]\rangle\}, D\rangle\end{array}}$$

(OBSERVABLE-FIELD-UPDATE)
$$\frac{\begin{array}{c}D = D' \uplus \mathcal{B}\langle \iota_b, \iota_a, f, O'\rangle \\ f(\iota_a) = O \uplus \{\mathcal{O}\langle \iota_o, F \uplus \{f := v'\}, M\rangle\} \\ f' = f[\iota_a \rightarrow O \cup \mathcal{O}\langle \iota_o, F \cup \{f := v\}, M\rangle] \\ D'' = D' \cup \mathcal{B}\langle \iota_b, \iota_a, f', O'\rangle\end{array}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\square[\iota_b.\iota_o.f := v]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[v]\rangle\}, D''\rangle\end{array}}$$

Figure A.21: Observable Domain Actor-Local Reduction Rules.

*Appendix A.4.3. Reduction Rules*

*Actor-local reductions.* We summarize the actor-local reduction rules in Figure A.21:

- PROCESS-MESSAGE: this rule replaces the rule for processing messages in Figure A.11. 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 $v$). The processing of an asynchronous message reduces to a synchronous method invocation followed by the runtime syntax commit. The most important change is that before the start of the turn, first a snapshot is taken of any domain which is not owned by the actor using the auxiliary *snapshot* function. While reducing the synchronous method invocation $\iota_a.\iota_o.m(\overline{v})$, any field access to an observable domain reference will be looked up using that snapshot.

- COMMIT: The runtime syntax commit is always the last expression that needs to be reduced before the end of a turn. The reduction of this rule replaces all the object heaps of the observable domains owned by the actor in $D$ with its own local copy using the auxiliary *commit* function.

- OBSERVABLE-INVOKE: In the case of method invocation, the method $m$ is looked up in the copy of the object that is located in the snapshot of the observable domain, $f(\iota_a)$. Note that the owner of the domain, $\iota_{a'}$, does not necessarily need to be the same as the actor that is invoking the method, $\iota_a$. Any actor can invoke a method on an observable domain reference, regardless of the owner of the domain. A method invocation reduces the method body expression $e$ with appropriate values for the parameters $\overline{x}$ and the pseudovariable `this`.

- OBSERVABLE-FIELD-ACCESS. Similar to method invocation, the field is looked up in the copy of the object that is located in the snapshot of the observable domain, $f(\iota_a)$. Any actor can access a field of an observable domain object.

- OBSERVABLE-FIELD-UPDATE. A field update to an observable domain reference can only be reduced if the owner of the observable domain is the same as the actor performing the update. Note that the field is only updated in the local snapshot that the actor has of the domain's heap. Any field updates are only propagated to the observable domain's heap at the end of a turn when the actor commits.

(NEW-OBSERVABLE-OBJECT)

$$\frac{\begin{array}{ccc}\iota_o \text{ fresh} & r = \iota_b.\iota_o & D = D' \cup \mathcal{B}\langle \iota_b, \iota_{a'}, f, O' \rangle \\ f(\iota_a) = O & f' = f[\iota_a \to O \cup \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\rangle] & \\ & D'' = D' \cup \mathcal{B}\langle \iota_b, \iota_{a'}, f', O' \rangle & \end{array}}{\begin{array}{c}\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{object}_{\iota_b}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D''\rangle\end{array}}$$

(NEW-OBSERVABLE-DOMAIN)

$$\frac{\begin{array}{cc}\iota_b, \iota_o \text{ fresh} & r = \iota_b.\iota_o \\ o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{[\![e]\!]_{\iota_b}\}}\rangle & \end{array}}{\begin{array}{c}\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{observable}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D \cup \{\mathcal{B}\langle \iota_b, \iota_a, \iota_a \to \{o\}, \{o\}\rangle\}\rangle\end{array}}$$

Figure A.22: Observable Creational Rules

*Rules for object and observable domain literals.* We summarize the immutable domain creation reduction rules in Figure A.22:

- NEW-OBSERVABLE-OBJECT: An object expression can only be reduced once its field initialize expressions have been reduced to a value. The effect of reducing an object literal expression is the addition of

a new object to the actor's local snapshot of the heap of the observable domain. The literal expression reduces to a domain reference $r$ to the new object.

- NEW-OBSERVABLE-DOMAIN: A domain literal will reduce to the construction of a new observable domain with the current actor, $\iota_a$ as its owner and with a single object in its heap. That domain is added to the set of domains in the configuration. Similarly to the rule for NEW-OBJECT, the observable domain expression can only be further reduced once its field initialize expressions have been reduced to a value. The domain expression reduces to an observable domain reference $r$ to the newly created object. The $[\![e]\!]_{\iota_d}$ syntax makes sure that all lexically nested object expressions are tagged with the domain id of the newly created domain.

$$\frac{\begin{array}{c}(\text{OBSERVABLE-ASYNCHRONOUS-SEND}) \\ \mathcal{B}\langle \iota_b, \iota_{a'}, f, O' \rangle \in D \\ A = A' \cup \{\mathcal{A}\langle \iota_{a'}, Q', e' \rangle\} \\ A'' = A' \cup \{\mathcal{A}\langle \iota_{a'}, \mathcal{M}\langle \iota_b.\iota_o, m, \overline{v} \rangle \cdot Q', e' \rangle\}\end{array}}{\begin{array}{l}\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_b.\iota_o \leftarrow m(\overline{v})] \rangle\}, D \rangle \\ \rightarrow_k \mathcal{K}\langle A'' \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\text{null}] \rangle\}, D \rangle\end{array}}$$

Figure A.23: Observable Asynchronous Message Reduction Rules

*Asynchronous communication reductions.* We summarize the asynchronous communication reduction rules in Figure A.23:

- OBSERVABLE-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at an observable reference. A new message is appended to the queue of the owner of the observable domain $\iota_{a'}$ (top part of the rule). The message send expression itself evaluates to `null`.

*Auxiliary functions.*

The auxiliary function *snapshot* ensures that at the start of a turn an actor takes a snapshot of each observable domain by storing a copy of its object heap. The first rule takes a new snapshot, $f[\iota_a \rightarrow O]$, of the latest version of the object heap, $O$, for any observable domain and recursively calls *snapshot* on the remaining domains. The second rule just returns the leftover set of domains, $D$, if all observable domains have been visited.

**Auxiliary functions**

$$\begin{array}{rcl} snapshot(\iota_a, D \cup \mathcal{B}\langle \iota_b, \iota_{a'}, f, O \rangle) & \stackrel{def}{=} & \mathcal{B}\langle \iota_b, \iota_{a'}, f[\iota_a \rightarrow O], O \rangle \cup snapshot(\iota_a, D) \\ snapshot(\iota_a, D) & \stackrel{def}{=} & D \quad \forall \iota_b : \mathcal{B}\langle \iota_b, \iota_{a'}, f, O \rangle \notin D \end{array}$$

$$\begin{array}{rcl} commit(\iota_a, D \cup \mathcal{B}\langle \iota_b, \iota_a, f, O \rangle) & \stackrel{def}{=} & \mathcal{B}\langle \iota_b, \iota_a, f, f(\iota_a) \rangle \cup commit(\iota_a, D) \\ commit(\iota_a, D) & \stackrel{def}{=} & D \quad \forall \iota_a : \mathcal{B}\langle \iota_b, \iota_{a'}, f, O \rangle \notin D \end{array}$$

The auxiliary function *commit* ensures that at the end of a turn any changes made to domains owned by the actor during that turn are committed. The first rule replaces the object heap, $O$, of any observable domain owned by the actor, $\iota_a$ with the snapshot stored in the semantic function, $f(\iota_a)$, and recursively calls *commit* on the remaining domains. The second rule just returns the leftover set of domains, $D$, if all observable domains owned by the actor have been visited.

*Appendix A.5. Shared Domains*

Objects owned by a shared domain can be accessed by any actor in the system given they have obtained a reference to that object. However, before an actor can read from and write to a shared domain object it first needs to obtain a view on the associated shared domain. A view is processed in its own turn and is called a notification. Adding shared domains does not change any of the existing reduction rules.

**Semantic Entities of Shacl**

| | | | |
|---|---|---|---|
| $D \subseteq$ **Domain** | ::= | $C \cup I \cup B \cup S$ | Domains |
| $S \subseteq$ **Shared** | ::= | $\mathcal{S}\langle \iota_s, l, S, E, R, O\rangle$ | Shared Domains |
| $R \subseteq$ **Request** | ::= | $\mathcal{R}\langle \iota_a, t, e\rangle$ | View Requests |
| $n \in N \subseteq$ **Notification** | ::= | $\mathcal{N}\langle \iota_d, t, e\rangle$ | Notifications |
| $Q \subseteq$ **Queue** | ::= | $\overline{m \mid n}$ | Queues |
| $v \in$ **Value** | ::= | $r \mid t \mid \text{null}$ | Values |
| $t \in$ **RequestType** | ::= | SH $\mid$ EX | Request Types |
| $l \in$ **AccessModifier** | ::= | R$(n)$ $\mid$ W $\mid$ F | Access Modifiers |
| $\iota_a \in S, E \subseteq$ **IsolatedId** | | | Actor Identifiers |
| $i \in \mathbb{N}$ | | | Integers |

Figure A.24: Additional semantic entities for shared domains

*Appendix A.5.1. Semantic Entities*

The set of domains is extended with the set of shared domains. A **Shared** domain has an identifier, $\iota_s$, a single **Access Modifier** $l$ (or lock), a set of actor ids, $S$, that currently have shared access to the domain and a set of actor ids, $E$, that currently have exclusive access to the domain. It also has a set of pending view requests, $R$, and its object heap, $O$. A pending **Request** has a reference $\iota_a$, to the actor that placed the request, the type of request, $t$, 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). 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 also extended to also allow the reception of notifications. The request-type is also a first class **Value**.

*Appendix A.5.2. Additional Syntax for Shared Domains*

New shared domains can be created using the `shared` literal. This creates a new object with the given fields and methods in a fresh shared domain. SHACL's `whenShared` and `whenExclusive` primitives are represented by the 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.

*Runtime syntax.* Additional runtime syntax was added to release a view at the end of a turn.

**Shacl Syntax**

**Syntax**

$e \in E \subseteq$ **Expression** ::= $\ldots \mid t \mid \text{shared}\{\overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid \text{acquire}_t(e)\{e\}$

**Runtime Syntax**

$e$ ::= $\ldots \mid \text{release}_t(\text{v})$

**Evaluation Contexts**

$e_\square$ ::= $\ldots \mid \text{acquire}_{e_\square}(e)\{e\} \mid \text{acquire}_v(e_\square)\{e\} \mid$
$\text{shared}\{\overline{f := v}, f := e_\square, \overline{f := e}, \overline{\text{m}(\overline{x})\{e\}}\}$

**Syntactic Sugar**

$\text{whenShared}(e)\{e'\} \stackrel{\text{def}}{=} \text{acquire}_{\text{SH}}(e)\{e'\}$
$\text{whenExclusive}(e)\{e'\} \stackrel{\text{def}}{=} \text{acquire}_{\text{EX}}(e)\{e'\}$

Figure A.25: Additional syntax for shared domains

*Evaluation contexts.* An additional evaluation context was added to define the order in which the expressions of the acquire primitive need to be reduced. Similar to the object syntax, the field initializer expressions of a shared domain expression need to be reduced to a value from left to right before the shared domain literal can be further reduced. An additional evaluation context was added to specify this behavior.

*Appendix A.5.3. Reduction Rules*

*Actor-local reductions.* We now summarize the actor-local reduction rules in Figure A.26:

- 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 ($\iota_a \in S \cup E$).
- FIELD-ACCESS, FIELD-UPDATE: 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 ($\iota_a \in S \cup E$) while field updates only apply in the case of an exclusive view ($\iota_a \in E$).

*Rules for object, domain and actor literals.* We summarize the creation reduction rules in Figure A.27:

- NEW-OBJECT: All object literals are tagged with the domain id of the lexically enclosing domain. The effect of evaluating an object literal expression is the addition of a new object to the heap of that domain. Evaluating an object literal reduces to a shared domain reference $r$ to the new object.
- NEW-SHARED-DOMAIN: A shared 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 shared domain expression can only be further reduced once its field initialize expressions have been reduced to a value. The domain expression reduces to a shared domain reference $r$ to the newly created object. $[\![e]\!]_{\iota_d}$ denotes

(INVOKE)

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

(FIELD-ACCESS)

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

(FIELD-UPDATE)

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

Figure A.26: Shared Domain Actor-Local Reduction Rules.

(NEW-OBJECT)

$$\frac{\iota_o \text{ fresh} \qquad r = \iota_s.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\rangle}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{object}_{\iota_s}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D \uplus \{\mathcal{S}\langle \iota_s, l, S, E, R, O\rangle\}\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[r]\rangle\}, D \cup \{\mathcal{S}\langle \iota_s, l, S, E, R, O \cup \{o\}\rangle\}\rangle\end{array}}$$

(NEW-SHARED-DOMAIN)

$$\frac{\iota_s, \iota_o \text{ fresh} \qquad r = \iota_s.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f := v}, \overline{\mathrm{m}(\overline{x})\{[\![e]\!]_{\iota_s}\}}\rangle \qquad D' = D \cup \{\mathcal{S}\langle \iota_s, \mathrm{F}, \emptyset, \emptyset, \emptyset, \{o\}\rangle\}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\mathrm{shared}\{\overline{f := v}, \overline{\mathrm{m}(\overline{x})\{e\}}\}]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\overline{r}]\rangle\}, D'\rangle\end{array}}$$

Figure A.27: Shared Creational Rules

a transformation that makes sure that all lexically nested object expressions are annotated with the domain id of the newly created shared domain. The access modifier is initially set to free. No actors have a view on the domain and the set of requests is empty.

*Asynchronous communication reductions.* We summarize the asynchronous communication reduction rules in Figure A.28:

- SHARED-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a shared domain reference. Reducing an asynchronous message to a shared 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`.

(SHARED-ASYNCHRONOUS-SEND)

$$\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_s.\iota_o \leftarrow m(\overline{v})]\rangle$$
$$\rightarrow_a \mathcal{A}\langle \iota_a, Q, e_\Box[\text{acquire}_{\text{EX}}(\iota_s.\iota_o)\{\iota_s.\iota_o.m(\overline{v})\}]\rangle$$

Figure A.28: Shared Asynchronous Message Reduction Rules

(ACQUIRE-VIEW)

$$\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\text{acquire}_t(\iota_s.\iota_o)\{e\}]\rangle\}, D \uplus \{\mathcal{S}\langle \iota_s, l, S, E, R, O\rangle\}\rangle$$
$$\rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\text{null}]\rangle\}, D \cup \{\mathcal{S}\langle \iota_s, l, S, E, R \cup \{\mathcal{R}\langle \iota_a, t, e\rangle\}, O\rangle\}\rangle$$

(PROCESS-VIEW-REQUEST)

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

(PROCESS-VIEW-NOTIFICATION)

$$\frac{D' = snapshot(\iota_a, D) \qquad D' = D'' \uplus \{\mathcal{S}\langle \iota_s, l, S, E, R, O\rangle\} \qquad S', E' = add(\iota_a, t, S, E) \qquad D''' = D'' \uplus \{\mathcal{S}\langle \iota_s, l, S', E', R, O\rangle\}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q \cdot \mathcal{N}\langle \iota_s, t, e\rangle, v\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e; \text{release}_t(\iota_s); \text{commit}\rangle\}, D'''\rangle\end{array}}$$

(RELEASE-VIEW)

$$\frac{l' = unlock(t, l) \qquad S', E' = subtract(\iota_d, t, S, E) \qquad D = D' \uplus \{\mathcal{S}\langle \iota_s, l, S, E, R, O\rangle\} \qquad D'' = D' \cup \{\mathcal{S}\langle \iota_s, l', S', E', R, O\rangle\}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\text{release}_t(\iota_s)]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[\text{null}]\rangle\}, D''\rangle\end{array}}$$

Figure A.29: Views reduction rules.

*View reductions.* We summarize the view reduction rules in Figure A.29:

- ACQUIRE-VIEW: This rule describes the reduction of `acquire` expressions. This rule simply adds the view-request to the set of requests in the domain. 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 is also considered as a turn of the actor. That means we first have to commit any changes to observable domains at the end of processing the notification. The request is removed from the set of requests and the access modifier of the domain is updated. 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 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 actor id, $\iota_a$, to the set of shared or exclusively accessible domains in the shared domain. 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 domain's set of available shared or exclusive views is updated according to the request using the *add* function. Processing a notification reduces to the the expression that is associated with the notification, followed by a release expression and a commit. Because processing a notification is regarded as a separate turn, the actor needs to take a snapshot of the latest observable domains and commit at the end of the turn (See Appendix A.4).

- RELEASE-VIEW: Releasing a view on the domain removes the actor id from the set of shared or exclusively views of the domain. 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, $E$, actor ids of a shared domain. The *add* rule adds actor ids to either sets while *subtract* rule subtracts actor ids from either sets, depending on the type of the view.

**Auxiliary functions and predicates**

$$
\begin{array}{rcl@{\qquad}rcl}
lock(\text{EX}, \text{F}) & \stackrel{def}{=} & \text{W} & unlock(\text{EX}, \text{W}) & \stackrel{def}{=} & \text{F} \\
lock(\text{SH}, \text{F}) & \stackrel{def}{=} & \text{R}(1) & unlock(\text{SH}, \text{R}(1)) & \stackrel{def}{=} & \text{F} \\
lock(\text{SH}, \text{R}(i)) & \stackrel{def}{=} & \text{R}(i+1) & unlock(\text{SH}, \text{R}(i)) & \stackrel{def}{=} & \text{R}(i-1) \\
\\
add(\iota_a, \text{EX}, \emptyset, \emptyset) & \stackrel{def}{=} & \emptyset, \{\iota_a\} & subtract(\iota_a, \text{EX}, \emptyset, \{\iota_a\}) & \stackrel{def}{=} & \emptyset, \emptyset \\
add(\iota_a, \text{SH}, S, \emptyset) & \stackrel{def}{=} & S \cup \{\iota_a\}, \emptyset & subtract(\iota_a, \text{SH}, S \cup \iota_a, \emptyset) & \stackrel{def}{=} & S, \emptyset
\end{array}
$$

*Appendix A.6. Differences Between* SHACL *and* SHACL-LITE

The operational semantics for SHACL-LITE was based on the semantics for the AmbientTalk language [25]. As such, we followed their syntax for many of the concepts of the basic event-loop model, which is different from the regular SHACL syntax. Semantically both languages are identical except for the fact that in SHACL-LITE an *object* expression is syntax that only allows the specification of the fields and methods of an object while in SHACL, `object` is a primitive with a single functional parameter that accepts any valid SHACL expression for initialization.

One difference between the actual implementation of the domain model and the operational semantics is that the implementation prioritizes exclusive view requests to prevent starvation. The operational semantics handles view requests non-deterministically.

*Appendix A.7. Conclusion*

We have presented an operational semantics for a key subset of the SHACL programming language. The operational semantics provides a formal account of SHACL actors, objects, synchronous and asynchronous communication and the four types of domains. What is novel about the semantics is that it generalises the concept of an object heap.