Vrije Universiteit Brussel

Faculty of Sciences and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Domains: Language Abstractions for Controlling Shared Mutable State in Actor Systems

Dissertation Submitted for the Degree of Doctor of Philosophy in Sciences

Joeri De Koster

Promotors: Prof. Dr. Theo D'Hondt
Dr. Tom Van Cutsem

January 2015

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / Fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 978 90 8224 807 4
NUR 989

ABSTRACT

Traditionally, concurrency models fall into two broad categories: flexible versus safe and manageable concurrency control. On one end of the spectrum there are shared-memory models such as threads and locks which are very flexible but offer almost no safety guarantees. On the other end of the spectrum are isolated message-passing models which are often more stringent, favoring safety and liveness guarantees.

The actor model is such a message-passing concurrency model and because of the asynchrony of its communication mechanism and isolation of its different processes the actor model avoids issues such as 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 trade-off is that the actor model sacrifices expressiveness with respect to parallel access to shared state.

This dissertation aims to show that strict isolation is not a prerequisite to guarantee deadlock and race condition freedom, and that breaking with no-shared-state concurrency can improve the flexibility of message-passing concurrency models while still maintaining their safety guarantees.

In this dissertation, we give an overview of the issues that come with representing shared resources in modern actor systems and then formulate an extension to the actor model that allows safe and expressive sharing of mutable state among otherwise isolated concurrent components. We propose *domains* as a set of novel language abstractions for encapsulating and sharing state. With the domain model, we introduce four types of domains, namely immutable, isolated, observable and shared domains. The design and implementation of the domain model is realized in the context of SHACL, a novel communicating event-loop actor language. We validate the usefulness of our model by applying it in practice through a case study in Scala.

SAMENVATTING

Traditioneel kunnen concurrency modellen opgedeeld worden in twee categorieën: flexibele versus veilig en beheersbare concurrencycontrol. Aan het ene uiteinde van het spectrum er zijn gedeeld-geheugen modellen zoals threads en locks die zeer flexibel zijn maar bijna geen veiligheidsgaranties bieden. Aan het andere uiteinde van het spectrum zijn geïsoleerde message-passing modellen die vaak strikter zijn, ten voordele van de veiligheidsgaranties van het resulterende model.

Het actormodel is zo een message-passing model en omdat het actormodel asynchrone communicatie hanteert en de verschillende processen strikt geïsoleerd zijn vermijdt dit model gekende concurrencyproblemen zoals deadlocks en elementaire data races. Dit vergemakkelijkt concurrent programmeren, voornamelijk in de context van complexe interactieve applicaties waarbij eigenschappen zoals modulariteit, veiligheid en fouttolerantie belangrijk zijn. De keerzijde van de medaille is dat dit model aan expressiviteit opoffert betreffende parallelle toegang tot gedeelde informatie.

Deze verhandeling stelt zich tot doel aan te tonen dat strikte isolatie van de verschillende actoren geen vereiste is om deadlocks en data races te vermijden. De breuk maken met *no-shared-state* parallelisme kan de flexibiliteit van het message-passing concurrency modellen verbeteren en tegelijkertijd hun veiligheidsgaranties bewaren.

In deze verhandeling starten we met een overzicht van de verschillende problemen die zich voordoen bij het voorstellen van gedeelde informatie in moderne actorsystemen. Nadien formuleren we een uitbreiding van het actormodel dat de programmeur in staat moet stellen om op een veilige en expressieve manier gedeelde informatie voor te stellen. We stellen *domeinen* voor als een set van verschillende taalabstracties voor het inkapselen en delen van informatie. Met dit model stellen we vier verschillende types van domeinen voor, met name niet-aanpasbare, geïsoleerde, observeerbare en gedeelde domeinen. Het ontwerp en de implementatie van domeinen zijn gerealiseerd in de context van SHACL, een nieuwe communicating event-loop actor programmeertaal. We valideren ook de bruikbaarheid van ons model door het toe te passen in de praktijk door middel van een case study in Scala.

ACKNOWLEDGEMENTS

Throughout my Phd I have been lucky enough to receive the support of not one, but two outstanding promotors. I would like to thank Prof. Theo D'Hondt for being largely responsible for shaping my views on programming language design and implementation. Every building needs a strong foundation and the work I did during my Phd is largely founded upon his principles. I would also like to thank Dr. Tom Van Cutsem for helping me put together the ideas presented in this dissertation. His involvement in my work truly marked the start of my Phd and many of the ideas and concepts introduced in this work derive from the discussions we had together. I cannot thank him enough for sticking around to see it through until the end.

I would also like to thank Dr. Stefan Marr, my favourite German I love to hate. Thank you for giving me the necessary whipping from time to time, those always led to peaks in my productivity and tremendously helped in getting some of my work published. I loved you as a fellow researcher and as an office mate and was sad to see you go.

A big thanks goes out to the members of my jury: Prof. Philipp Haller, Dr. Roel Wuyts, Prof. Wolfgang De Meuter, Prof. Jennifer Sartor, Prof. Philippe Cara, Dr. Yann-Michaël De Hauwere. Thanks for critically reading my dissertation and helping in improving the quality of this text.

I would like to thank all the (former) members of the Software Languages Lab for providing an excellent environment to work in. Special thanks goes out to the Parallel Programming People for all the insightful debates an discussions during our meetings.

I would also like to thank my family and friends to take my mind off "dinges met computers" and do some more fun stuff from time to time.

Last, but not least, I would like to thank Florence Dusart, my best friend, life partner, wife and mother of my child. We met when I started studying Computer Science and she has since supported me through my entire academic career. All of the important choices I made in life, including doing the Phd of which this work is the end result, I made together with her. I am therefore deeply indebted to her.

This work is funded by a Phd scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

INHOUDSOPGAVE

LIJST VAN FIGUREN

LIJST VAN TABELLEN

LIST OF LISTINGS

1 INTRODUCTION

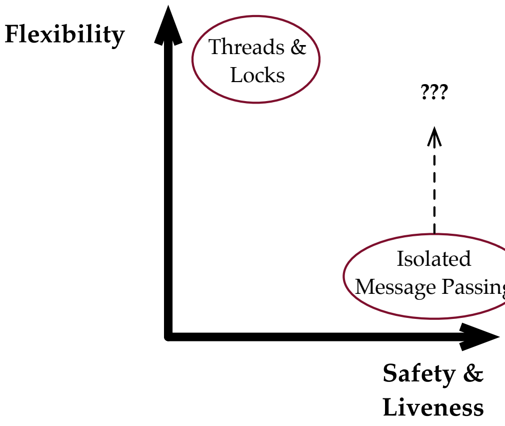
Moore's law [Moore, 1965] states that the density of transistors in central processing units (CPUs) doubles approximately every two years. In the past decades, software developers have relied on this "free lunch" for performance gains of their applications without necessarily releasing new versions of their software. Unfortunately, we have hit a performance wall and the free lunch is over [Sutter, 2005]. For now, Moore's Law seems to hold. However, the exponential increase of the number of transistors does not translate into an increased clock speed for a single CPU. Instead, hardware manufacturers are investing in running multiple CPUs in a single chip. In the past few years we have seen a transition towards integrating multi-core processors in commodity hardware such as smartphones, laptops, and workstations. This has led to an increased interest in concurrent programming models. In the past concurrent programming had largely remained a tool for specialist developers in the field of embedded systems or high performance computing. However, today, concurrent programming is also becoming an important factor for exploiting these new multi-core architectures in mainstream desktop, mobile and web applications.

Exploiting parallelism in the context of high performance computing is often analogous to parallelizing algorithms that are composed of executing a set of homogeneous tasks. Data-parallel programming models such as Fork/Join [Blumofe et al., 1995], MapReduce [Dean and Ghemawat, 2008] and Dataflow [Ashcroft and Wadge, 1977] have traditionally been well suited for exploiting that kind of parallelism. The focus of these models is more on abstractions for parallel programming rather than manually introducing concurrency. However, interactive applications are generally more complex and the different components of such applications often map onto heterogeneous tasks. This means that developers would benefit from a concurrency mechanism that is well suited to model each component or task as a parallel software entity. Concurrency mechanisms that fit this category include threads and locks, software transactional memory (STM) [Shavit and Touitou, 1995], communicating sequential processes (CSP) [Hoare, 1978] and actors [Hewitt et al., 1973]. This does not mean that the degree of parallelism should be limited to the number of components of an application. Inside a single component it is possible to find opportunities to parallelize a set of homogeneous tasks using the various data-parallel programming models. What it does mean is that the governing concurrency mechanism for interactive applications should be able to safely model heterogeneous tasks in a modular, reusable, secure, and fault-tolerant way.

In a larger context, the focus of this dissertation is about the contrast between flexible concurrency control and manageable concurrency control by guaranteeing safety and liveness. These two sets of requirements are fundamentally conflicting and designing a new concurrency mechanism is often an act of choosing between one or the other. On one end of the spectrum there are models such as threads that are very flexible but inherently non-deterministic. This non-determinism can cause safety issues such as race conditions and pruning that non-determinism with locks can cause liveness issues such as deadlocks [Lee, 2006]. On the other end of the spectrum there are isolated message passing models such as CSP and Actors which follow a strict "share nothing" approach, thus favoring safety and liveness guarantees. The downside of these models is that they are often more stringent when it comes to accessing shared state. Fig. 1.1 graphically sketches this trade-off.

The starting hypothesis of this dissertation is that developers of complex interactive applications benefit most from a model with high safety and liveness guarantees. However, this dissertation aims to show that this strict isolation is not always necessary to provide those guarantees and that breaking with no-shared-state concurrency can improve flexibility while still maintaining deadlock and data-race freedom.

While the ideas presented in this dissertation could be applied to other concurrency models (see Sec. 9.3), this dissertation focusses on the actor model. In the actor model, applications are decomposed into concurrently running actors. Actors are isolated (i. e., they have no direct access to each other's state), but may interact via (asynchronous) message passing. While it is often used as a distributed programming model, actors remain equally useful as a higher-level alternative to multithreading in shared-memory architectures. Both component-



Figuur 1.1.: The trade-off between flexible and performant concurrency control and manageable concurrency control

based and service-oriented architectures can be modeled naturally using actors. In this dissertation, we will 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.

1.1. Problem Statement

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 often strictly enforce the isolation of actors: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. Examples of pure actor languages include Erlang [Armstrong et al., 1996], SALSA [Varela and Agha, 2001], E [Miller et al., 2005], AmbientTalk [Van Cutsem et al., 2007], and Kilim [Srinivasan and Mycroft, 2008]. 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 very often added to existing languages whose concurrency model is based on shared-memory multithreading. For Java alone, examples include ActorFoundry [Astley, 1998-99], Actor Architecture [Jang, 2004], ProActive [Baduel et al., 2006], AsyncObjects [Plotnikov, 2007], JavAct [J.-P. Arcangeli, 2008], Jetlang [Rettig, 2008-09], and AJ [Zwicky, 2008]. Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, such as Scala Actors [Haller and Odersky, 2007] and Akka [Allen, 2013].

What 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 [Haller and Odersky, 2010]. 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.

On the one hand, 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. On the other hand, impure actor libraries are more flexible at the cost of some of those safety guarantees.

1.2. Research Vision

The goal of this work is to enable safe and expressive state sharing among actors in pure actor languages. To achieve this goal, we aim to relax the strictness of pure actor languages 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.

While the primary focus of this dissertation is to enable safe and expressive state sharing among actors, there is also a note to be made on efficiency. Today, multicore hardware is the prevalent computing platform, both on the client and the server sides [Sutter, 2011]. While multiple isolated actors can be executed perfectly in parallel by different hardware threads, shared access to a single actor can still form a serious sequential bottleneck. In pure actor languages, shared mutable state is modeled with a specific actor and all requests sent to it are serialized, even if some requests could be processed in parallel, e. g., requests to simply read or query part of the actor's state. Pure actors lack multiple-reader, single-writer access, which is required to enable truly parallel reads of shared state.

1.3. Contributions

This dissertation makes the following contributions in the field of concurrent programming language design and more specifically, the actor model.

Actor System History We give an overview of some of the important contributions in the field of actor programming. We illustrate that the different actor systems can be classified along four distinct actor families. We give an overview of some of the most important properties of each actor system. The semantics of the original actor model ensure that processing a message happens in a single isolated step. This is an important principle for formal reasoning as well as application development, as the amount of processing done in a single step can be made as large or as small as necessary. We formally define this principle as the *isolated turn principle*.

SHACL, a Communicating Event-loop Actor Language The SHACL (**Sh**ared **act**or **l**anguage) programming language is an imperative prototype-based object-oriented programming language that adopts the communicating event-loop actor model as its model for concurrency. SHACL is designed as a platform for experimenting with new language abstractions for coordinating access to shared mutable state in the communicating event-loop actor model.

The Domain Model The core contribution of this dissertation is the domain model. Domains are a family of language abstractions for controlling shared state in an actor system. We present four types of domains, namely immutable, isolated, observable, and shared domains and show how they enable safe and expressive access to shared state while preserving the safety guarantees of the original actor model.

Communicating Event-Loop Calculus We defined an operational semantics for a subset of the SHACL programming language. This operational semantics serves as a reference specification for the semantics of our language abstractions regarding domains. By unifying object heaps of the original model and isolated domains we show that they are overlapping language concepts. Our calculus serves as an extensible basis for experimenting with semantics for various other types of domains.

A survey of shared-state in practical actor systems We performed a survey of an existing set of relevant open-source Scala projects that employ actors for concurrency control. This survey illustrates what synchronization mechanisms programmers of modern actor systems currently employ to synchronize access to a shared resource. We establish that programmers mix different synchronization mechanisms depending on the desired application-specific properties. Based on this survey we illustrate how the identified synchronization mechanisms could be translated to the domain model and validate that the desired properties are still guaranteed.

1.4. Supporting Publications and Technical Contributions

Parts of this dissertation's contributions have been published. This section discusses these publications briefly to highlight their relevance to this work.

Published Papers: Shared and Observable Domains
The core contribution of this dissertation is *The Domain Model* as discussed in Chapter 5. The following papers are about the design and rationale behind the different domains.

In the following works we explored the core semantic features of shared domains:

- Joeri De Koster, Stefan Marr, and Theo D'Hondt. Synchronization views for event-loop actors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 317–318, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145873. URL http://doi.acm.org/10.1145/2145816.2145873

- Joeri De Koster, Tom Van Cutsem, and Theo D'Hondt. Domains: Safe sharing among actors. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! '12, pages 11–22, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414644. URL http://doi.acm.org/10.1145/2414639.2414644

This work was extended and formalized in the following journal paper:

- Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. Domains: Safe sharing among actors. *Science of Computer Programming*, 98, Part 2(0):140 – 158, 2015. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2014.02.008. URL http://www.sciencedirect.com/science/article/pii/S0167642314000495. Special Issue on Programming Based on Actors, Agents and Decentralized Control

The design and rationale behind observable domains has been explored in the following work:

- Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. Tanks: Multiple reader, single writer actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 61–68, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541331. URL http://doi.acm.org/10.1145/2541329.2541331

Technical Contibutions
This dissertation is also supported by a number of technical contributions which are listed here.
The operational semantics for the domain model is discussed in Chapter 6. This

operational semantics serves as a precise specification of the semantics of the domain model. A standalone version can be found here:

- Joeri De Koster and Tom Van Cutsem. Shacl: Operational semantics. Technical report, Vrije Universiteit Brussel, 2013. `http://soft.vub.ac.be/Publications/2013/vub-soft-tr-13-26.pdf`

The SHACL language was designed as a research platform to experiment with synchronization mechanisms for the communicating event-loop actor model. A full implementation can be found here:

- Joeri De Koster. The shacl programming language. `http://soft.vub.ac.be/~jdekoste/shacl/`, 2014

1.5. Dissertation Outline

The rest of this dissertation is organized as follows.

Chapter 2 gives an overview of the history of Actor Systems, starting from its inception by Hewitt et al. [1973] until the present day (November 2014). This chapter distinguishes the *style* of actor model from each Actor System according to four major families and catalogues them according to their properties.

Chapter 3 argues that there is a lack of good abstractions to represent shared resources in modern actor systems by giving an overview of the currently available state of the art techniques.

Chapter 4 gives an in-depth overview of the communicating event-loop actor model. The communicating event-loop actor model is an interesting model for structuring large interactive applications where actors function mostly as components of the system. We show that this model has a number of desirable properties over other actor models for that purpose. This chapter also introduces the SHACL programming language. SHACL is designed as a platform for experimenting with new language abstractions for representing shared resources in the communicating event-loop actor model. The chapter ends by illustrating the limitations of the communicating event-loop model specifically when it comes to representing shared resources.

Chapter 5 proposes the main contribution of this dissertation, *The Domain Model*. This chapter 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.

Chapter 6 gives a formal account of the domain model by providing an operational semantics for a significant subset of the SHACL programming language. The aim of the operational semantics is to serve as a reference specification of the semantics of the language abstractions regarding domains.

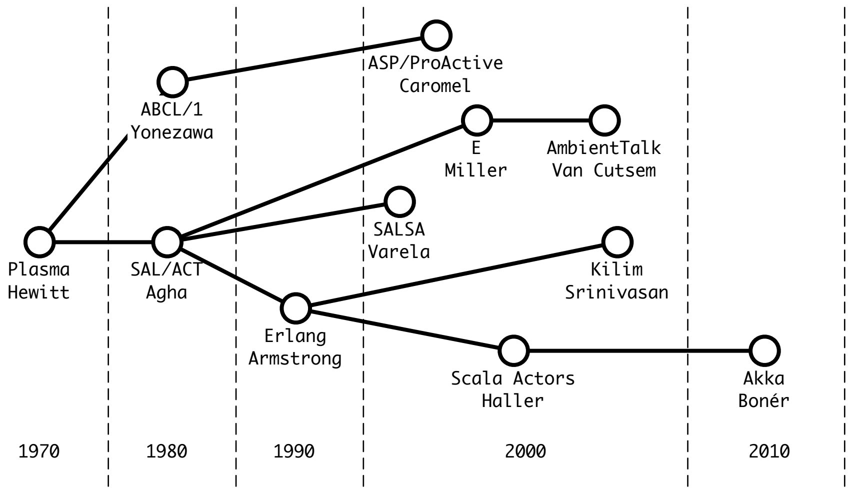
Chapter 7 presents *domain handlers* as a possible strategy for implementing the domain model. A domain handler defines a number of traps that serve as callbacks for the interpreter. Each domain handler can then specify the behavior of a certain domain type by specializing the default trap(s).

Chapter 8 validates the usefulness of the domain model. We start with a survey of synchronization patterns used by developers in existing open-source Scala projects. We then show that these examples can be translated to the domain model.

Chapter 9 concludes this dissertation and highlights avenues for future work.

2
CONTEXT: ACTOR SYSTEMS

The actor model was originally proposed by [Hewitt et al., 1973] in the context of artificial intelligence research at MIT. The original goal was to have a programming model for safely exploiting concurrency in distributed workstations. In December 1975, in an attempt to understand the actor model described by Hewitt, [Sussman and Steele, 1975] wrote a continuation based interpreter for a Lisp-like language called Scheme. They came to the conclusion that Hewitt's "actors" were very similar to scheme lambda expressions and had their roots in the lambda calculus [Church, 1936]. In a purely functional context, sending a message is very similar to passing a continuation. Later, [Agha, 1986, 1990] revised this model and made actors stateful by allowing them to change their behavior. This way of doing state updates has an important advantage over conventional assignment statements as this severely coarsens the granularity of operations that need to be considered when analyzing a system. This insight is, for this dissertation, perhaps the most important contribution of his work. In an actor model, the granularity of reasoning is at the level of the processing of a single message. This means that processing a message can be regarded as being done in a single step. Throughout the rest of this dissertation we refer to this principle as **the isolated turn principle**. Agha's actor model consists of three basic actor primitives: *create*, *send* and *become*. These three primitives spawned a host of different actor systems and can still be found in modern actor languages and libraries today. This chapter gives an overview of the ancestry of actor systems from 1975 until today (2014) as seen in Fig. 2.1. Afterwards we classify the different actor languages into four major paradigms and categorize them in terms of the properties they provide.



Figuur 2.1.: A selection of actor languages and their ancestry

2.1. The History of Actor Systems

The actor model was originally proposed by Hewitt et al. [1973] as a computational model for artificial intelligence. It was meant for modeling parallel communication based problem solvers. In October of 1975 Hewitt and Smith [1975] wrote a primer on a language called PLASMA, the first language implementation of the actor model. In PLASMA, actors consist of a behavior and an inbox. A behavior specifies the set of messages accepted by an actor and the code that should be executed to handle each type of message. The inbox of an actor holds all incoming messages. An actor perpetually takes one message from its inbox and processes that message. Actors communicate with each other via message passing which consists of sending a request from one actor (called the messenger) to another actor (called the target). The request and the messenger are packaged as an envelope and put into the inbox of the target actor (`request: message; reply-to: messenger`). Given that envelope, the behavior of the target actor then specifies how the computation continues with respect to the request. The messenger is typically used as the reply address to which a reply to the request should be sent. The simplest control structure that uses this request-reply pattern in most programming languages is the procedure call and return. A recursive implementation of *factorial* written in PLASMA is given in Lst. 2.1.

In this example factorial is defined to be an actor of which the behavior matches the requests of incoming envelopes with one element which will be

```
(factorial ≡
  (≡> [=n]
    (rules n
      (≡> 1
            1)
      (≡> (> 1)
            (n * (factorial <= (n - 1)))))))
```

Listing 2.1: Factorial function written in PLASMA.

called `n`. The rules for `n` are, if it is `1`, then we send back `1` to the messenger of the envelope. Note that this is done implicitly. If it is greater than `1`, we send a message to the factorial actor to recursively compute the factorial of `(n - 1)`.

2.1.1. Agha's Actor Model: ACT, SAL and Rosette

Agha [1986, 1990] proposes a variation on the actor model as a concurrent object-oriented programming model. The main focus was to produce a platform for distributed problem solving in networked workstations. In his model concurrent objects, i. e. actors, are self-contained, independent components that interact with each other by asynchronous message passing. In his work he presents three basic actor primitives:

- `create`: Creates an actor from a behavior description. Returns the address of the newly created actor.
- `send`: Asynchronously sends a message from one actor to another by using the address of the receiver. Immediately returns and returns nothing.
- `become`: Allows actors to replace their behavior. The next message that will be received by that actor is processed by the new behavior.

These three primitives are the basic building blocks for most actor systems today and have been very influential in the development of any actor language that follows this work. The sequential subset of the model he describes is mostly functional. Any state changes are specified by replacing the behavior of an actor. This has an important advantage over conventional assignment statements as this severely coarsens the granularity of side-effecting operations that need to be considered when analyzing a system. On the one hand, an actor can only change its own behavior, meaning that actors are fully isolated. On the other hand, changing the behavior of an actor only comes into effect when processing the next message. This means that, the processing of a single message can be regarded as a single isolated operation. This mechanism allows state updates to be aggregated into a single become statement and significantly reduces the amount of control flow dependencies between statements. The example in Lst. 2.2 is written in the Rosette actor language [Tomlinson et al., 1988] which was based on this model.

```
(define Cell
  (mutable [content]
    [put [newcontent]
      (become Cell newcontent)]
    [get
      (return 'got-content content)]))

(define my-cell (create Cell 0))
(get my-cell)
```

Listing 2.2: An actor in Rosette.

The `mutable` form is used to create an *actor generator* that is bound to `Cell`. That generator can be used with the `create` form to create an instance of that actor. Each actor instance has its own inbox and behavior. Following the keyword mutable is a sequence of identifiers that specify the mutable fields of that actor. In our example, any `Cell` actor will have one mutable field, namely the content of that cell. After that is a specification of all the methods that are understood by the actor. A method is specified by a keyword followed by a table of arguments. In this case the `put` method expects a value for the new content. Afterwards follows the body that specifies how each method should be processed. If one wishes to modify the state of a mutable field one can use the `become` form to replace the behavior of an actor using the actor generator. The `return` form is used to send back the result of a computation to the implicit sender of the original message.

There are many library implementations of the actor model based on [Agha, 1990] for different languages such as Smalltalk (Actalk [Briot, 1989]) and C++ (ACT++ [Kafura, 1990], Broadway [Sturman and Agha, 1994] and Thal [Kim, 1997]).

2.1.2. ABCL/1

Around the same time [Yonezawa et al., 1986] worked on a object-oriented concurrent programming language called ABCL/1. In this language, each object has its own processing power and may have its own local persistent memory. In this model state changes are not specified in terms of behavior updates (become) but rather by individual assignment statements. However, the state of each active object is isolated from one another. This means that state updates are also isolated and because messages are processed entirely sequentially the isolated turn principle still holds[1].

```
[object Cell
  (state [contents := nil])
  (script
    (=> [:put newContent]
      contents := newContent)
    (=> [:get] @ From
      From <= contents))]

Cell <= [:get]
```

Listing 2.3: An active object in ABCL/1.

There are three types of messages in ABCL/1: *past*, *now* and *future*. *Past type* messages are sent to the receiver and immediately return. The sender does not wait for the receiver to process the message before continuing its current computation. This is the default message type we knew from other actor models up till now. *Past type* messages can be sent by using the following syntactical form: `[T <= M]`. Where `M` is the message and `T` is the receiver of the message. This type of message is called "past" because sending a message finishes before it is processed by the receiver object. *Now type* messages are very similar to procedure call and return. When an object `O` sends a *now type* message to another object `T`, `O` will wait for `T` to process that message and send back a result before continuing with its current computation. *Now type* messages can be sent by using the following syntactical form: `[T <== M]`. While *now type* messages provide a convenient synchronization mechanism for concurrent activities, it can cause potential deadlocks as it is a blocking operation. *Future type* messages are used when the sender of a message does not need the result of the message im-

[1]Barring the use of *express messages*, which can potentially interrupt the processing of a message and thus violate this principle.

mediately. *Future type* messages can be sent by using the following syntactical form: `[T <= M $ x]` where `x` is a variable that will bind a *future object*. That future can be used by the sender of the message to wait for the result. Trying to reference `x` will cause the computation to block until the return value of the message becomes available. In other actor models, the sender of a request, `O`, has to finish its computation before being able to receive the response from the receiver, `T`. If sending this request and processing the result is part of O's task, this often leads to an unnatural breakdown of that task in different execution steps. ABCL/1's futures was the first attempt to solve that issue.

2.1.3. Erlang

Erlang [Armstrong et al., 1996] was the first industrial language to adopt the actor model as its model of concurrency. It was developed at the Ericsson and Ellemtel Computer Science Laboratories as a declarative language for programming large industrial telecommunications switching systems. While in style Erlang has many similarities with Agha's actor model, in Erlang, actors are not represented by a behavior. Rather actors are represented by processes that run from start to completion. Erlang actors can use the primitive `receive` to specify what messages an actor understands. When evaluating a `receive` expression the actor pauses until a message is received. If a message is received, the matching code is evaluated and execution continues until a new `receive` block is evaluated. One can use recursion to ensure that an actor continues processing incoming messages. This is illustrated by the following example:

```
loop(Contents) ->
  receive
    {put, NewContent} ->
      loop(NewContent);
    {get, From} ->
      From ! Contents,
      loop(Contents)
  end.

MyCell = spawn(loop, [nil]).
MyCell ! {get, self()}.
```

Listing 2.4: An Erlang process.

We create a new actor process by using the primitive `spawn`. This will call the provided function, `loop`, in a new process and returns that process' process id.

The cell uses the primitive `receive` to match incoming get- and put-messages. Once the message body is processed we recursively call the loop with our updated state.

2.1.4. SALSA

SALSA [Varela and Agha, 2001] was one of the first actor languages implemented on top of Java. The implementation translates SALSA code into Java code that can be compiled together with the SALSA actor library to Java byte-code with any Java compiler and run on any JVM. SALSA was proposed as an actor-based language for mobile and internet computing. A few of the main contributions are three new language mechanisms to help coordinate asynchronous communication between different actors. When an actor sends an asynchronous message to another actor, that actor may include an implicit *customer* to which the result should be sent after the message has been processed. This can be done by using one of three kinds of continuations, namely token-passing continuations, join continuations and first-class continuations. The example in Lst. 2.5 illustrates how to use actors and token-passing continuations in SALSA.

```
behavior Cell {
  Object contents;
  Cell(Object initialContents){
    contents = initialContents;
  }
  void put(Object newContents) {
    contents = newContents;
  }
  Object get() {
    return contents;
  }
}

Cell myCell = new Cell(null);
myCell<-get() @ standardOutput<-print(token);
```

Listing 2.5: A SALSA behavior.

Similar to ABCL/1 state updates are not specified in terms of behavior updates (become) but rather by individual assignments. The SALSA compiler ensures actors are correctly encapsulated by serializing any object that is transmitted from one actor to another. Thus, the isolated turn principle holds. The last line in the example illustrates how to use token continuations in SALSA. When the `myCell` actor is done processing the `get` message the result can be printed on the screen. We do this by sending a `print` message to the standard output using `token` as a value. `token` is a special keyword that evaluates to the value provided by the last token-passing continuation. This abstraction is very similar in use compared with ABCL/1's future type messages. The main difference being that using a future in ABCL/1 is a blocking operation while a token-passing continuation postpones the evaluation of the continuation until the result of the message send becomes available. Meanwhile other messages can be processed.

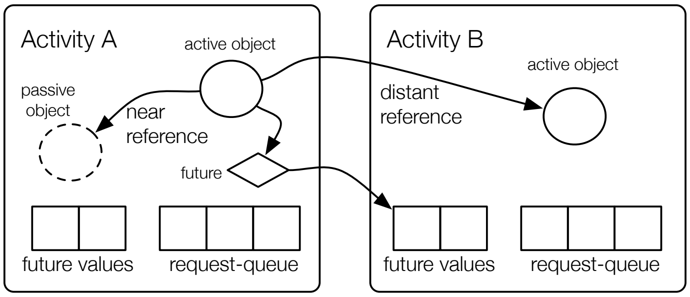
Join continuations in SALSA, as illustrated by the example below are a way to group several messages. Any token continuation following a join continuation will wait until all the messages are processed. Once this is done the token continuation receives an array with the tokens produced by the different messages.

```
join(author<-writeChapter(1),
     otherAuthor<-writeChapter(2))
  @ editor<-review(token) @ publisher<-print(token);
```

SALSA also has first-class continuations. The keyword `currentContinuation` can be used as a token continuation to delegate computation to the context in which the message processing is taking place. For example, in the example above, the editor can potentially send messages to other actors during its review. Any results produced by those messages can be delegated to the publisher by passing `currentContinuation` as a token continuation.

2.1.5. Asynchronous Sequential Processes and ProActive

Asynchronous Sequential Processes [Caromel et al., 2009] is a programming model similar to ABCL/1 in that it is a concurrent object-oriented programming model. However, contrary to ABCL/1, not every object in this model is an active object. Rather, actors in this model are represented by an *activity*. Each activity has a single root object called the *active* object. Different activities do not share memory, the active objects' whole object graph is deep-copied into the activity and the copied objects are then called passive objects. Every activity always has a single active object and potentially many passive objects. Any method call on an active object will result in an asynchronous request being sent to the active object and returns a future. The request is stored in a request-queue and is called *pending*. Later this request will be *served* and when it is finished the request is *calculated* and the future is replaced with a (deep) copy of the return value.



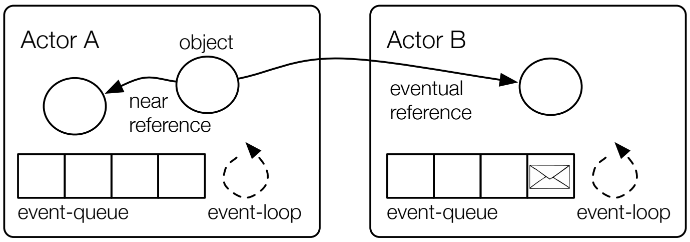
Figuur 2.2.: Asynchronous Sequential Processes

Fig. 2.2 illustrates two activities, each with their own active object. Any method call via a distant reference will add that call to the request-queue of the corresponding activity and result in a future. The result of the method call is stored in a 'store' of future values. Similarly to ABCL/1's futures, execution will block if any attempt is made to perform a strict operation (e. g., a method call) on such a future. Execution resumes when the corresponding request is calculated. Isolation of the different activities is guaranteed by passing passive objects by copy between the different activities. All references to passive objects are always local to an activity and any method call on a passive object is synchronously executed. An implementation of this model can be found in ProActive [Baduel et al., 2006], which is a framework for Java.

2.1.6. E Programming Language

The E programming language [Miller et al., 2005] was the first language to introduce *the communicating event-loop actor model*. This model takes a very similar approach to Asynchronous Sequential Processes with the exception that it does not make a distinction between passive and active objects. In this model, each actor is represented as a *vat*. A 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 are owned by exactly one vat. Within a vat, references to objects owned by that same vat are called *near references*. References to objects owned by other vats are called *eventual references* (see Fig. 2.3).

The type of reference determines the access capabilities of that vat's thread of execution on the referenced object. Generally, actors are introduced to one



Figuur 2.3.: The communicating event-loop model

another by exchanging addresses. In the communicating event-loop model such an address is always in the form of an eventual reference to a specific object. The referenced object then defines how another actor can interface with that actor. The main difference between communicating event-loops (CEL) and other actor languages up to this point was that other actor languages usually only provide a single entry point or address to an actor. A CEL can define multiple objects that all share the same message queue and event-loop and hand out different references to those objects. The example in Lst. 2.6 illustrates how to create an object in E and send it an eventual message `get`.

```
def cell {
  var contents := null
  to put(newContents) {
    contents := newContents
  }
  to get() {
    return contents
  }
}

var promise := cell<-get()
when (promise) -> {
  println(promise)
}
```

Listing 2.6: A vat in E.

When you send an eventual message to an object the message is enqueued in the event queue of the owner of the object and immediately returns a *promise*. That promise will be resolved with the return value of the message once that message is processed. It is not allowed to use a promise as a near reference. If you want to make an immediate call on the value represented by a promise, like printing it on the screen, you must set up an action to occur when the promise resolves. This is done by using the `when` primitive. When the promise for the value of the `get` message becomes resolved, the body of the `when` primitive is executed. During that execution the promise is resolved and can be used as a local object.

The communicating event-loop model was later adopted by AmbientTalk [Van Cutsem et al., 2007], a distributed object-oriented programming language which has been designed for developing applications on mobile ad hoc networks. AmbientTalk was designed as an *ambient-oriented programming* (AmOP) language [Dedecker et al., 2006]. It adds to the actor model a number of new primitives to handle disconnecting and reconnecting nodes in a network where connections are volatile. The core concurrency model however remains faithful to the original communicating event-loops of E.

2.1.7. Scala Actor Library and Akka

The Scala Actor Library [Haller and Odersky, 2007] offers a full-fledged implementation of Erlang-style actors on top of Scala. Today, the Scala Actor Library remains one of the most widespread used actor implementations. Partly because Scala is built on top of the JVM and thus Scala seamlessly interoperates with code written in Java. Scala Actors can use two different primitives to receive a message. On the one hand, `receive` suspends the current actor together with its full stack frame until a message is received. Once the message is received the actor can continue processing that message and the context in which the `receive` block was executed is not lost. On the other hand, `react` suspends the actor with just a continuation closure. This closure only contains information on how to proceed with processing an incoming message. The context in which the `react` was called is lost. This type of message processing has the benefit of being more lightweight and thus scales to a larger number of actors. The example in Lst. 2.7 illustrates how to create a new actor in Scala and send it a message.

The Akka [Allen, 2013] actor library is an actor-framework for Scala. It stays more faithful to the original Agha actor model in the sense that an Akka actor has a single global message handler to process incoming messages. Instead of invoking `receive` to process incoming messages you implement a `receive` method that defines how all incoming messages should be processed. To change the global message handler (i. e., the behavior) of an actor in Akka one can use the `become` control structure.

```
class Cell[T](var contents: T) extends Actor {
  def act() {
    loop {
      react {
        case Get =>
          sender ! contents
        case Put(newContents: T) =>
          contents = newContents
      }
    }
  }
}

val cell = new Cell(0)
cell.start
cell ! Get
```

Listing 2.7: An actor in Scala.

```
class Cell[T](var contents: T) extends Actor {
  def receive = {
    case Get =>
      sender ! contents
    case Put(newContents: T) =>
      contents = newContents
  }
}

val system = ActorSystem("MySystem")
val cell = system.actorOf(Props(new Cell(0)), name = "cell")
cell ! Get
```

Listing 2.8: An actor in Akka.

2.1.8. Kilim

Kilim [Srinivasan and Mycroft, 2008] is an actor framework for Java. The Kilim weaver post-processes Java byte-code to add a lightweight implementation of actors and provide strong isolation guarantees. The example in Lst. 2.9 illustrates how to implement a `Cell` actor in Kilim. Each actor class needs to specify an `execute` method as entry point for the actor. Getting a message from a mailbox is a blocking operation and the Kilim weaver makes sure that context switching is possible during the execution of any method that is annotated with the `@pausable` annotation. Objects that are transmitted over a mailbox have to implement the `Message` interface and are passed by copy between the different actors. This is the standard technique used to ensure isolation between the different processes. However, there exist extensions to Kilim to do zero-copy message passing [Gruber and Boyer, 2013].

```
class Cell extends Actor {
  Mailbox<Msg> mb;
  private int contents = 0;
  Cell(Mailbox<Msg> mb) { this.mb = mb; }

  @pausable
  public void execute() {
    while(true) {
      Msg m = mb.get();
      if (m.type == "Get") {
        m.replymb.put(contents);
      }
      if (m.type == "Put") {
        contents = m.contents
      }
    }
  }
}

Mailbox<Msg> outmb = new Mailbox<Msg>();
new Cell(outmb).start()
Mailbox<Msg> replymb = new Mailbox<Msg>();
outmb.put(new Msg(replymb, 42, "Put"));
Msg reply = replymb.get();
```

Listing 2.9: An actor in Kilim.

2.2. Actor System Classification and Properties

In Sec. 2.1 we have shown some of the most prominent implementations of the actor model. While each of those actor systems is different, we can distinguish the *style* of actor model according to four major families. In this section we will classify each of the discussed languages according to those four main families. Each actor system in a single family has a number of common features that are shared between all actor systems in that family. However, a number of other features are still actor system dependent. Tab. 2.1 on page 27 gives an overview of the different distinct features of each actor system. The goal of this classification is to give a complete view on all the different actor systems and their properties. This will allow us to derive and extract some general properties and guarantees for each actor system.

2.2.1. Classification of Actor Systems

This section gives an overview of the four major families according to which we classified each actor system. The four major families are: Original actor model, processes, active objects and communicating event-loops.

2.2.1.1. Original Actor Model

Actor systems in this category implement the original model proposed by Agha. All actor systems in this category provide the actor model by means of three basic primitives, namely **create**, **send** and **become**. The sequential subset of languages in this category is typically mostly functional (e. g., ACT, SAL, Rosette). Changes to the state of an actor are aggregated in a single `become` statement. The only (implicitly) mutable data-structures are the inbox and the behavior of an actor. Isolation of the different actors is guaranteed because an actor can only change its own behavior and that change is only observable between the processing of messages. The Akka actor library also belongs in this category as it also implements the three basic actor primitives. However, its sequential subset is built on top of the imperative language Scala, meaning that actor isolation is not guaranteed.

2.2.1.2. Processes

The Erlang implementation of the Actor Model was unique as it did not specify an actor's implementation as a reentrant behavior. Rather, to represent actors, it employs processes that run from start to completion. Any actor system that represents an actor as a process falls into this category. Once the process terminates, the actor stops and can no longer receive any messages. To receive a message, an actor has to evaluate a `receive` statement. Once the message has been processed, execution can continue where it left off. The sequential subset of Erlang is functional and similar to Agha's model, changes to the state of an actor are typically aggregated between the different `receive` statements. Erlang actors are fully isolated and thus the isolated turn principle holds[2]. However, in this case, the notion of a "turn" is not so easily defined. It makes sense to regard all operations between the processing of two consecutive `receive` statements as being processed in a single isolated operation or turn.

The Scala Actor Library and Kilim also belong in this category as they also represent actors as processes that run from start to completion. However, in the case of Scala, actor isolation is not guaranteed. Kilim does guarantee actor isolation at compile time by ensuring that any object that crosses actor boundaries does not have any internal aliases.

2.2.1.3. Active Objects

ABCL/1 moved away from the classical representation of actors and looked at actors from an object-oriented perspective. Actors are represented by objects with their own thread of control and isolated memory. The main difference with the classical representation of actors is that state updates in this model are not aggregated in a single `become` operation. In fact, the behavior of an actor is fixed in this model: it is the interface of the active object. Rather than changing their behavior, actors can modify individual instance variables. Asynchronous Sequential Processes was an adaptation of this model in which an extra distinction is made between *active* and *passive* objects. Active objects are passed by reference and can only be communicated with via asynchronous message passing. Passive objects are passed by copy to ensure isolation and can be communicated with via synchronous method invocation. SALSA is another language that falls into this category. In SALSA, each behavior specifies an active object and passive objects are copied via serialization when crossing actor boundaries. Javascript webworkers is another implementation of Active Objects. Any object that crosses webworker boundaries is copied by serialization.

[2]Erlang does allow a limited form of shared state between different actors in the form of Erlang Term Storage (ETS) tables. These are not part of the core language and will not be discussed here.

2.2.1.4. Communicating Event-Loops

The communicating event-loop model was first introduced by E. In this model actors are represented by `vats`. The main difference between event-loops and actor models based on active objects is that the communicating event-loop model does not make a distinction between active and passive objects. In this model every object is a passive object that is owned by a vat. A vat can then hand out references to those different objects and other vats can use those references to asynchronously communicate with that vat. In contrast with other actor models, a vat does not have a single behavior as entry point. Rather, every reference to an object defines a different behavior that can be the target of an asynchronous message. To ensure isolation objects are not passed by copy but rather by proxy. AmbientTalk also adopted the communicating event-loop model.

2.2.2. Actor Properties

Each of the four families discussed gives some indication of the properties of the actor system. However, these properties still remain largely dependent on the specific implementation of the actor system. In this section we give an overview of all the features and properties we use to classify the different actor systems discussed in Sec. 2.1. Tab. 2.1 gives an overview of all actor systems and their properties. There are four main classes of features and properties. First we look at how each system **processes individual messages**. Secondly, we look at **how messages are received** by the actor. Thirdly, we look at what mutable state is available in the actor system and how the actor system handles **state changes**. Lastly, we classify the different actor systems according to the **granularity in which actors are meant to be used** on a single node.

2.2.2.1. Message Processing

An important part of any actor system is the way in which messages are processed. This is typically the sequential subset of the language. An important sidenote here is that, any property that holds for the sequential subset of the language, typically only holds for the processing of a single message. For example, any actor system that upholds the isolated turn principle guarantees that each message is processed sequentially and fully isolated. However, once you broaden that boundary to the processing of several messages, most of these properties no longer hold. In this section we only concern ourself with properties that hold during the processing of a single message.

	Message Processing			Message Reception				State Changes		Actors per Node
	Paradigm	Continuous	Consecutive	Interface	Flexibility	# Interfaces	Order	Granularity	Isolation	Granularity
Original Actor Model										
Agha (ACT, SAL, Rosette)	Functional	Continuous	Consecutive	Behavior	Flexible	1	Unordered	Aggregated	Isolated	Fine-grained
Akka	Imperative	Blocking	Consecutive	Behavior	Flexible	1	Unordered	Individual	Shared-Memory	Fine-grained
Processes										
Erlang	Functional	Continuous	Consecutive	Receive	Flexible	1	Unordered	Aggregated	Isolated	Fine-grained
Scala Actor Library	Imperative	Blocking	Consecutive	Receive	Flexible	1	Unordered	Individual	Shared-Memory	Fine-grained
Kilim	Imperative	Blocking	Consecutive	Mailbox	Flexible	1	FIFO	Individual	Isolated	Fine-grained
Active Objects										
ABCL/1	Imperative	Blocking	Interleaved	Methods	Fixed	1	FIFO	Individual	Isolated	Coarse-grained
ASP (ProActive)	Imperative	Blocking	Consecutive	Methods	Fixed	1	FIFO	Individual	Isolated	Coarse-grained
SALSA	Imperative	Continuous	Consecutive	Methods	Fixed	1	FIFO	Individual	Isolated	Coarse-grained
Communicating Event-Loops										
E	Imperative	Continuous	Consecutive	Methods	Fixed	*	E-ORDER	Individual	Isolated	Coarse-grained
AmbientTalk	Imperative	Continuous	Consecutive	Methods	Fixed	*	FIFO	Individual	Isolated	Coarse-grained

Tabel 2.1.: Actor Languages Classification and Properties

Paradigm The sequential subset of an actor language can either be functional or imperative. If it is functional then, typically, the only way to modify state is to change the behavior of the actor. If it is imperative then that means that extra care needs to be taken to guarantee isolation of the different actors. If the isolated turn principle is guaranteed, then the choice of paradigm does not impact the concurrency properties of the resulting model.

Continuous The sequential subset of a language can allow blocking statements or can ensure a continuous processing of each message. In the latter case actors are guaranteed to process a message from start to completion without having to worry about deadlocks. Again, this only applies to the processing of a single message. Other forms of lost progress can still occur between the processing of different messages.

Consecutive We consider a message to be processed consecutive if it is processed from start to completion without being interleaved with the processing of other messages. This is usually guaranteed unless there is some way to interrupt the processing of a single message (e. g. express messages in ABCL/1).

2.2.2.2. Message Reception

Incoming messages are always stored in the inbox of an actor. How those messages are taken out of that inbox can differ between the different actor systems. In this section we discuss some properties of actor systems according to how they take messages out of their inbox before processing them.

Interface The interface (i. e. behavior) of an actor can be specified in various ways. Some actor systems specify a behavior from a behavior description. Others use special primitives such as `receive` to take a message from their inbox. Others use an object-oriented style where the interface of the actor corresponds to the interface of an object and a message send corresponds to a method invocation.

Flexibility The interface to an actor can be fixed or flexible. When the interface of an actor is fixed that means that actor will always understand the same set of messages at any given point in time. When the interface is flexible, the set of messages an actor understands can vary over time. This is typically done by changing the behavior of the actor. However, this does not imply that changing the behavior of an actor has to somehow change its interface. A behavior is state-full and an actor can change its behavior to update its internal state without changing what messages it understands. Similarly, having a fixed interface does not imply that actors always respond to a message in the same way. With an imperative sequential subset is still possible to change how an actor responds to a message depending on earlier state updates.

Number of interfaces Traditionally, every actor has a single entry point, namely the interface (i. e., the behavior) of that actor. However, in the case of the communicating event-loop model, each actor can hand out many references to its own object and each reference can specify a different interface. Because this is the most distinguishing property of the communicating event-loop model we wanted to give this property its own section here.

In order In the case of a fixed interface, it makes sense to process messages in the same order they arrived in the inbox of the actor. However, when the interface is flexible it can be opportune to process messages out of order depending on what messages are supported by the behavior that is in place at the start of each turn.

2.2.2.3. State Changes

Regardless of whether their sequential subset is functional or not, all implementations of the actor model have some form of mutable state (e. g. the behavior/inbox of an actor).

Granularity State changes can be aggregated or on an individual, i. e. per variable, basis. If the sequential subset of the actor language is functional then the only mutable state of the actor is typically its behavior.

Isolation Isolation is guaranteed when no two actors can read-write or write-write to the same memory location. In actor systems where the sequential subset is functional this is guaranteed because in those languages the only mutable state is the behavior of an actor and actors are only able to modify their own behavior. In actor systems where the sequential subset is imperative some extra care needs to be taken when sharing mutable state. For example, by (deep) copying any data structure when it crosses actor boundaries.

2.2.2.4. Actors Per Node

The original intention for the actor model was to provide a programming model for expressing concurrent programs over different nodes in a distributed network. The message passing model and isolation of the different actors is a good fit for such systems. As such, most actor systems include support for distribution. However, where they do differentiate is how actors were meant to be used on a single node. This ranges from Erlang, which is known for its lightweight implementation of actors and supposed to run many actors on a single node, to AmbientTalk, which is an actor language designed to deploy only a few actors on each node in a peer-to-peer network with volatile connections.

2.3. The Isolated Turn Principle

The semantics of the original actor model enable a *macro-step semantics* [Agha et al., 1997]. 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 dissertation 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.

To satisfy this principle, an actor system must satisfy the following three properties from Tab. 2.1:

Continuous message processing. The processing of a message cannot contain any blocking operations. Any message is always entirely processed from start to finish. Because of this property, processing a single message is free of deadlocks.

Consecutive message processing. An actor can only process messages from its own inbox and those messages can only be processed one by one. The processing of a message cannot be interleaved with the processing of other messages of the same actor.

Isolation. The state of any actor is fully encapsulated. This property is mainly guaranteed by adopting a *no-shared-state* policy between actors. Any object that is transmitted across actor boundaries is either copied, proxied or immutable. This property ensures that the processing of a single message in the actor model is free of low-level data races.

The isolated turn principle guarantees that the actor model is free of low-level data races and deadlocks. However, on the one hand, 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 potential bad interleavings and ensure correct execution. On the other hand, high-level deadlocks can still occur when actors are waiting on each other to send a message before progress can be made.

2.4. Conclusion

The goal of this chapter was to give an overview of the current state of the art in actor systems. This section discusses each actor system with its properties and classifies them according to the four major families we have identified. Tab. 2.1 gives an overview of all the described actor systems and their properties.

3

SHARED STATE IN MODERN ACTOR SYSTEMS

This chapter discusses the different strategies to share state in modern actor systems. As we saw in Chapter 2, the actor model is either made available via dedicated programming languages (actor languages), or via libraries in existing languages (actor libraries). Ensuring isolation of the different actors can typically only be achieved in a language approach. As such, actor languages are often *pure*[1], in the sense that they strictly enforce actor isolation: the state of an actor is fully encapsulated, cannot leak, and asynchronous access to it is enforced. The major benefit of pure actor systems is that the developer benefits from strong safety guarantees: low-level data races are ruled out by design. The main drawback and the strongest limitation of pure actor systems is that direct resource sharing between actors is completely ruled out. Instead, approaches such as *delegate actors* or *replication* are used to achieve shared state depending on whether consistency or parallelism are the main concern. Actor libraries on the other hand are often *impure*[2], they do not strictly enforce actor isolation. It is easy for a developer to use the underlying shared-memory concurrency model as an "escape hatch" when direct sharing of state is the most natural or most efficient solution. However, once the developer chooses to go this route, the benefits of the high-level actor model are lost, and the developer typically has to resort to manual locking to prevent data races. This chapter discusses the different ways to represent a shared resource in both pure and impure actor systems.

[1]A notable exception being Erlang when using Term Storage Tables [Armstrong et al., 1996]

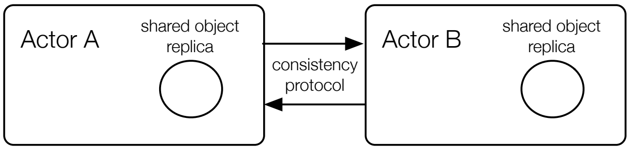
[2]A notable exception being SALSA [Varela and Agha, 2001], which guarantees isolation by using a compiler technique

3.1. Shared State in Pure Actor Systems

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. In this section we discuss the disadvantages of both approaches.

3.1.1. Replication

One option for representing a shared resource in the actor model is to replicate this resource inside different actors that require access to it. Fig. 3.1 illustrates two actors that share a single object by replicating it over the different actors. Each actor has direct read access to the replicated object. Writes have to be propagated to the different replicas by means of a consistency protocol. This approach has disadvantages with regard to consistency, memory overhead, and performance:



Figuur 3.1.: Replicating a shared object over the different actors

Consistency Even though we only consider shared memory systems, the CAP theorem [Brewer, 2000] still applies. If we want to increase the availability of our shared state, we need to lower its consistency. Unfortunately, whether lowering the consistency is possible is entirely application dependent. Moreover, keeping replicated state consistent requires a consistency protocol that usually does not scale well with the number of participants. The amount of interactions needed between the different actors grows exponentially with the number of participants. In our specific example this approach can be used, but if we consider applications with hundreds of components this is no longer feasible. In general, keeping replicated state consistent is a hard problem.

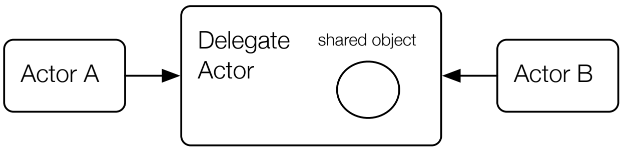
Memory usage Replication increases the memory usage with the amount of shared state and the number of actors. Depending on the granularity with which actors are created, this might incur a memory overhead that is too high.

Copying cost In certain applications, short lived objects need to be shared between different actors and the cost of copying them is greater than the cost of the operation that needs to be performed on them. For example, calculating the sum of all the elements in a vector in parallel would be less efficient than the sequential version if we first need to copy each subset of the vector to a different actor.

The isolated turn principle guarantees that during any single turn an actor has a consistent view over the whole state space. Keeping this guarantee in the case of state replication would involve some communication between the actors containing the replicated state upon each update of that state. In conclusion, offering state replication as an optimisation strategy for certain applications as a caching technique can be useful when eventual consistency is enough. However, when designing a general purpose actor system, usually a stricter consistency model is required.

3.1.2. Delegate Actor

The more natural solution for representing shared state in pure actor systems is by encapsulating that shared state in a separate delegate actor. Any actor that wants to access the shared resource is forced to use asynchronous message passing.



Figuur 3.2.: Using a delegate actor

Using a delegate actor to encapsulate a shared resource does not require any consistency protocols and thus scales better than replication because the amount of communication needed to synchronize access to the shared resource only grows linearly with the number of participants. There are however four different classes of problems when using this approach: Code fragmentation and continuation-passing style enforced, read-access to the shared resource cannot be parallelized and message-level deadlocks and race conditions can occur.

In this section we will discuss these four issues using the example in Lst. 3.1 written in Rosette. On line 1 we define a mutable cell as an actor behavior 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 then a `set` message to increment that value by one.

```
(define Cell
  (mutable [content]
    [put [newcontent]
      (become Cell newcontent)]
    [get
      (return 'reply content)]))

(define cell (create Cell 0))
```

```
(define Client
  (behavior
    [start [cell]
      (send cell 'get)]
    [reply [content]
      (return 'put (+ content 1))]))

(define client (create Client))
(send client 'start cell)
```

Listing 3.1: Rosette: A shared counter represented by a delegate actor

3.1.2.1. 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), also known as programming without a call stack [Hohpe, 2006]. The problem with this style of programming is that it leads to “inversion of control” [Johnson and Foote, 1988].
In the example in Lst. 3.1 we see that every time the client wants to retrieve the value of the mutable cell, the continuation of our program has to be modeled in a different callback message, namely `reply`. 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. The code that is responsible for incrementing the value of our cell is fragmented over the different callbacks.

Future-type messages Many actor languages, especially in the Active Object and Communicating Event-Loop family of 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 var promise := cell<-get()
2 when (promise) -> {
3   cell<-put(promise + 1)
4 }
```

Listing 3.2: A future-type message in E

In E, every asynchronous message send is a future-type message send and returns a promise (i. e. future). Lst. 3.2 illustrates the use of future-type messages in E. We use the return value of the message on line 1 to store a reference to a promise in the `promise` variable. Once the `get` message is processed by the vat that owns the `cell` object, that promise will be resolved. On line 2, the `when` primitive is used to register a closure that needs to be called when the promise 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, violating the condition that messages have to be processed continuously. On the other hand, when the future is resolved, the registered closure has to be called during its own turn. That turn cannot be interleaved with other turns of the same actor. Otherwise we would violate the condition that messages have to be processed consecutively.

In E, registering a closure using the `when` 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 promise is resolved an event that is responsible for calling the closure is scheduled in the event queue of the vat 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. There are even techniques to prevent the use of a CPS transformation (e. g., C# and Scala `assync` and `await`). 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.

SALSA's token-passing continuations are very similar to futures except that the CPS transformation is done by the compiler. In ABCL, accessing a future is a blocking operation that is done implicitly. This violates the continuous message processing condition that is needed for satisfying the isolated turn principle. In Scala, by default, the registered closure is processed in parallel with other messages of the same actor. This violates the consecutive message processing condition that is needed for satisfying the isolated turn principle. There are ways for the programmer to circumvent this issue in Scala by using custom execution contexts to guarantee that the future is executed on the same worker thread as the actor.

Processes. The family of actor languages that use processes to represent actors have the distinct advantage of being able to receive messages anywhere during the control flow of the execution of the actor. Thus, waiting for a reply can be done without breaking the control flow of the program. Continuation passing style is not enforced and the code is not fragmented. However, several messages still need to be processed in order to complete the operation, we cannot guarantee the isolated turn principle for the whole operation.

3.1.2.2. No Parallel Reads

State that is conceptually shared 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. Most of current actor systems do not support this optimization, even though it is completely safe to execute read operations in parallel. [Scholliers et al., 2014] proposes an optimization where messages that are tagged read-only can be processed in parallel.

3.1.2.3. 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 Lst. 3.1, the asynchronous messages sent by the `client` on lines 13 and 15 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 or she 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.

Flexible behavior. In actor systems where the behavior of an actor is flexible, (i. e. original actor model and processes) we can put extra synchronization constraints on the order in which the `Cell` actor has to process incoming messages by changing the behavior of that actor between messages.

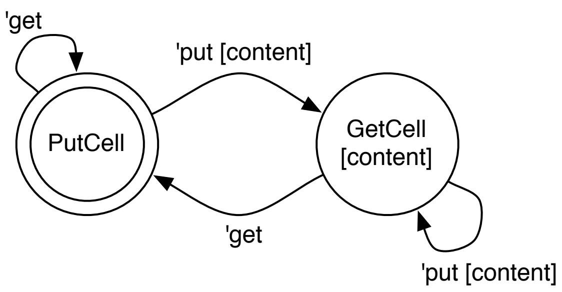
```
(define GetCell
  (mutable [content]
    [get
      (become PutCell)
      (return 'reply content)]))
```

```
(define PutCell
  (behavior
    [put [newcontent]
      (become GetCell newcontent)]))
```

Listing 3.3: Rosette: A finite state machine

Lst. 3.3 illustrates how we can turn an actor into the finite state machine shown in Fig. 3.3 by restricting what messages it understands each turn. Only the first `get` message will be processed. All other `get` messages are blocked in the inbox of the `Cell` actor until a `put` message is processed. However, there are two issues with this approach. On the one hand, if one actor does not follow this protocol and sends only a `get` message all other messages are blocked and a message-level deadlock occurs (See Sec. 3.1.2.4). On the other hand, by restricting the behavior of an actor we also restrict the potential parallelism. If an actor offers different services for which the different messages can be interleaved, we effectively restrict the parallelism.

Batch messages. [Yonezawa et al., 1986] proposes batch messages to circumvent the lack of synchronization. A batch message groups several messages in a single batch that are processed consecutively by the receiver. However, batch messages do not solve the synchronization problem in the case when there are



Figuur 3.3.: Flexible behavior: Finite state machine

data dependencies between the different messages. For instance in our example, the value passed to the put message depends on the return value of the get method.

3.1.2.4. Message-level Deadlocks

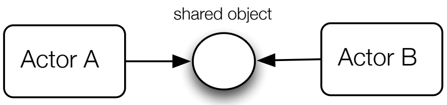
Changing the behavior of an actor forces that actor to prioritize the processing of certain messages. If the reception of those messages depends on progress made by that actor a deadlock can occur. Relying on callbacks can have a similar effect when the invocation of different callbacks are dependent on one another.

3.1.2.5. Conclusion

In most pure actor systems delegate actors are the common idiom to represent a shared resource. However, in this section we have shown that using a delegate actor forces any computation on that state to span several turns. As we saw in Sec. 2.3, the guarantees given by the isolated turn principle only apply during a single turn. Common concurrency problems such as deadlocks and race conditions can still occur when an operation spans several turns. 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.

3.2. Shared State in Impure Actor Systems

Actor libraries are often added to an existing language whose concurrency model is based on shared-memory multithreading. When opting for a library approach it is often difficult to enforce isolation of the different actors. As such, actor libraries are often impure. Scala, which inherits shared-memory multithreading as its standard concurrency model from Java, features multiple actor frameworks, namely Scala Actors [Haller and Odersky, 2007] and Akka [Allen, 2013]. What these libraries have in common is that they do not enforce actor isolation, i. e., they do not guarantee that actors do not share mutable state. The upside is that developers can easily use the underlying shared-memory concurrency model as an "escape hatch", when direct sharing of state is the most natural or most efficient solution. However, once the developer chooses to go this route, the benefits of the high-level actor model are lost, and the developer typically has to resort to manual locking to prevent data races.



Figuur 3.4.: Both actors have direct access to the shared object.

Fig. 3.4 illustrates two actors that have direct shared access to a shared object. In that case, another synchronization mechanism such as locks or Software Transactional Memory (STM) needs to be provided. Typically, if these synchronization mechanisms are provided, they are not well integrated with the actor model and thus compromise the isolated turn principle. In this section we will discuss a number of issues with these synchronization mechanisms when they are combined with the actor model.

3.2.1. Locks

Locks can be used as a synchronization mechanism to protect concurrent access to shared state. However, in practice, locks often only yield understandable programs for very simple interactions [Lee, 2006]. Acquiring nested locks can lead to a situation where different actors are stalled while attempting to acquire a lock that the other holds, thus causing a deadlock. Each actor that is involved in the deadlock is unable to process other incoming messages and any further actor-local progress is lost.
Another problematic case is when the actor library has a lightweight implementation of actors where many actors are scheduled on fewer worker threads. In that case, the provided locking mechanism is not always well integrated with the actor framework. A worker that stalls on acquiring a lock is unable to make progress which can lead to a multitude of problems. On the one hand, it can potentially compromise fairness, which is an implicit assumption of most actor

systems. On the other, if all workers threads are involved in acquiring a lock, global progress can be lost even without there inherently being a deadlock. For example, if all workers are stalled on acquiring a lock that is held by an actor that is currently not being processed. A know technique to solve that last problem is by temporarily extending the pool of workers when all workers are stalled [Haller and Odersky, 2009].
Furthermore, acquiring a lock is a blocking operation which compromises the isolated turn principle.

3.2.2. Software Transactional Memory

STM is another mechanism that can be used to synchronize access to shared state. However, integrating an STM in an actor system is also not always trivial because of the *abort-and-retry* mechanic that is inherent to transactional memory systems. Any operation that is executed within a transaction needs to be idempotent and this leads to several problems when the STM is not well integrated with the actor model. On the one hand, any message that is sent during a transaction needs to be either part of the transaction or postponed to be sent when the transaction commits. Otherwise messages can be sent multiple times when a transaction is executed multiple times. On the other hand, any mutable state that is local to the actor needs to be part of the transactional memory. As transactional memory incurs a certain overhead, ideally we would like to only pay the cost for the actually shared state.

3.3. Conclusion

Impure actor systems are flexible when it comes to representing shared resources. The programmer can access the underlying shared-memory model to represent a shared resource. Unfortunately, this flexibility compromises on the overall safety guarantees of the original model. Pure actor systems are on the other end of the spectrum, they are more stringent. They follow a no-shared-state concurrency model and shared resources have to be represented through either replication or a delegate actor. The benefit of these restrictions being that pure actor systems can offer a lot of guarantees to the programmer, the most important one being the isolated turn principle. However, the isolated turn principle only provides some guarantees for the processing of individual messages. Concurrency issues such as race conditions and deadlocks remain an issue when an operation spans several messages, which is often the case when accessing a shared resource in a pure actor system as the programmer is forced to employ asynchronous communication to access that resource. Ideally we would want a representation for a shared resource that allows us to access that resource synchronously for the duration of a single turn without compromising the isolated turn principle.

4 COMMUNICATING EVENT-LOOPS

The Communicating Event-Loop (CEL) model was originally intended as an object-oriented programming model for secure distributed computing [Miller et al., 2005]. However, the starting hypothesis of this dissertation is that this model is also useful in a shared memory context. Modularity, reusability, security, and fault-tolerance are prime qualities required from the implementation platform when modeling component-based software and plug-in architectures. We claim that the communicating event-loop model, as first instantiated in E, offers these qualities and for the remainder of this dissertation, we consider it as the foundation on which we will build abstractions for the safe sharing of mutable state between otherwise isolated actors. On the one hand, the object-oriented sequential subset of the communicating event-loop model allows designing modular, reusable software systems, because it provides a standard object-oriented programming model without any distractions related to concurrency. On the other hand, its actor model offers fault-tolerance by isolating the different actors. Any actor that crashes due to algorithmic faults, logic or network failures does not have an impact on other concurrently running actors. This chapter motivates the choice for the communicating event-loop model. Subsequently, SHACL, a communicating event-loop actor language is introduced. The SHACL virtual machine was developed as a platform for experimenting with new language features. SHACL is implemented as a new language rather than an extension of an existing language as it allows us to enforce certain properties of our model. The sequential subset of SHACL is a prototype-based object-oriented programming model while communicating event-loop actors are offered as the abstraction for introducing concurrency.

4.1. Why Communicating Event-Loops?

In Chapter 2 we have already shown that generally actor models are free of deadlocks and data races which are useful properties in a shared memory context. However, specifically for implementing component-based software and plug-in architectures, the CEL model has a number of properties which make it highly suitable for that task. This section motivates that reasoning. On the one hand, a *vat* (i. e. actor) is a coarse-grained concurrency mechanism that naturally fits with each component or plug-in. Actor isolation allows for a fault-tolerant implementation (See Sec. 4.1.1). On the other hand, the fact that a vat can have multiple public behaviors allows for a modular implementation (See Sec. 4.1.2).

4.1.1. Fine-grained versus Coarse-grained Concurrency

In component-based software, an application is composed of a number of components and each of those components offers a number of services. When introducing concurrency, rather than parallelizing individual components or services, it makes sense to adopt a more coarse-grained approach by running each component in a separate process. That way, services offered by different components can be executed in parallel. This idea does not fit with all actor models. On the one hand, the intent of the *original actor model* was to offer lightweight actors as an abstraction such that they can be used as fine-grained building blocks. Languages in the *process* category implement actors in the same vein. The Erlang virtual machine even makes a statement of being able to run thousands of hybrid processes in parallel [Armstrong et al., 1996]. The CEL model on the other hand promotes actors as a more coarse-grained concurrency mechanism. Each *vat* (i. e. actor) is a concurrent container for many objects. Distinct actors are strongly isolated, which allows for a fault-tolerant implementation.

4.1.2. Flexible versus Fixed Behavior

The benefit of actor models that allow behavior modification is that changing the behavior effectively postpones the execution of some messages. This allows the programmer to put extra synchronization constraints on the order in which messages are processed and effectively allows the programmer to use actors as finite state machines (See Sec. 3.1.2.4). However, if another client actor does not follow the correct protocol when using a service of our component, that might lead that actor to be stuck in a state and effectively cause a deadlock.

There are various reasons why the client would not respect the ordering of messages (e. g. the client is malicious software or the client has simply crashed). This is not a desirable property when designing robust fault-tolerant software. The advantage of actor models with a fixed behavior is that they can guarantee that any message will eventually be processed. This also means that the processing of a message cannot be postponed, it either succeeds or fails. The drawback is that extra care needs to be taken to ensure that messages from different sources are executed in the correct order. An added benefit of using the CEL model is that an actor in that model is not limited to a single behavior. For example, we could have a component modeled as an actor that wants to export two services. Each service implements a certain interface that can be exported to the clients of the component. There are two ways to combine two services in an actor model where actors have a single behavior. Either one would have to define a new behavior which has an interface that is a combination of both services; or define a separate protocol so the client can specify which service he wants to address first. In the CEL model, a component that provides different services can export each of those services as an *eventual reference* to an object that implements the interface to that service. Because each actor in a communicating event-loop model can have many behaviors, those interfaces can be cleanly separated.

```
def reservations := # ...

def clientService {
  to printTicket(clientID) {
    printer.printTicket(reservations.find(clientID))
  }
  to makeReservation(clientID, flightID) {
    def reservation := newReservation(clientID, flightID)
    reservations.add(clientID, reservation)
    return reservation
  }
}

def managerService {
  to printFlightReservations(flightID) {
    printer.print(reservations.summary())
  }
}
```

Listing 4.1: Flight reservation component written in E.

Lst. 4.1 illustrates how to export different behaviors as interfaces to different services in E. The example defines two services, one for clients to make a reservation and print out their tickets and one for managers to print out a summary of all reservations. Both services make use of the same shared state, namely the `reservations` database. For this reason it makes sense to put them both in the same vat. Any state update of the `reservations` database is automatically synchronized by the event queue of the actor. An added benefit of separating both services in different interfaces is that it is possible to cleanly separate which clients can access what service. Only managers that have obtained an *eventual reference* to the `managerService` can access that service.

4.2. SHACL: A Communicating Event-Loop Language

The SHACL (**Sh**ared **ac**tor **l**anguage) programming language is designed as a platform for experimenting with new language abstractions for representing shared resources in the actor model. This section gives an informal overview of SHACL minus the domains abstractions. See Chapter 6 for a full operational semantics of a subset of SHACL.

4.2.1. Imperative Programming in SHACL

The sequential subset of SHACL's programming model is based on that of Pico [De Meuter et al., 1999] which is a Scheme-like [Adams et al., 1998] language designed for teaching but with a C-like [Kernighan, 1988] syntax.

Primitive values. SHACL has a number of atomic values such as booleans, integers, floats, strings. The only composite value that is offered by default in the imperative subset of the language are tables, which are arrays indexed starting from index 1.

Syntax. The SHACL syntax for the imperative subset of the language is explained by means of the three by three matrix depicted in Tab. 4.1.

	variable	**table**	**function**
reference	$\mathtt{nam}$	$\mathtt{exp}[\mathtt{exp_1}]$	$\mathtt{exp}(\mathtt{exp_1},\ldots,\mathtt{exp_n})$
definition	$\mathtt{nam} : \mathtt{exp}$	$\mathtt{nam}[\mathtt{exp_1}] : \mathtt{exp_2}$	$\mathtt{nam}(\mathtt{exp_1},\ldots,\mathtt{exp_n}) : \mathtt{exp}$
assignment	$\mathtt{nam} := \mathtt{exp}$	$\mathtt{exp}[\mathtt{exp_1}] := \mathtt{exp_2}$	$\mathtt{nam}(\mathtt{exp_1},\ldots,\mathtt{exp_n}) := \mathtt{exp}$

Tabel 4.1.: SHACL basic syntax

This two-dimensional matrix results from merging all possible combinations of two design decision dimensions. In SHACL, an expression is always evaluated with respect to a certain environment. Each row in the table provides syntax on how to manipulate this environment. To refer to something in the environment, to add something to the environment (with `:`) and to change something in the environment (with `:=`). Each column in the table gives syntax to manipulate the three basic imperative SHACL values, namely variables, tables and functions. We can refer to variables, define them with an initial value or update them. Likewise we can refer to an element of a table at a certain position, we can define a new table with a certain size and we can update an element of a table. Finally, we can call a function, define a new function or update the body of a function with a new one.

```
1  fac(n):
2    if(n < 2,
3       1,
4       n * fac(n - 1));
5
6  x: 5;
7
8  fac(x);
```

Listing 4.2: Factorial definition in SHACL

The example in Lst. 4.2 illustrates how to define the recursive factorial function in SHACL. On line 1 a new function `fac` is defined with a single parameter, `n`. Its body consists of a single SHACL expression, namely a call to a function called `if`.There is no additional syntax for special forms. A conditional `if` expression is written down in the same way as a functional application with three arguments, namely the predicate, the consequent and alternative. On line 6 a new variable, `x`, is defined and on line 8 the factorial function is called with a single argument, `x`.

Parameter passing. When the formal parameter of a function is an ordinary identifier, SHACL follows standard call-by-value semantics. Atomic values are sent by copy while tables and functions are sent by reference. However, in SHACL and Pico, the formal parameter expression of a function can also be an application invocation (i. e. call-by-name, see Lst. 4.3). In that case, the formal parameter is bound to a new closure that closes over the environment of the call-site.

```
1 foreach(f(x), t):
2   for(i:1, i<=t.size(), i:=i+1,
3       f(t[i]));
4
5 foreach(
6   if(x % 2 == 0,
7     display("Even", eoln),
8     display("Odd", eoln)),
9   [1, 2, 3]);
```

Listing 4.3: Call-by-name parameter in SHACL

In Lst. 4.3, a new function `foreach` is defined with two formal parameters: an application expression `f(x)` and an ordinary identifier `t`. When calling foreach on line 5, the first expression that is passed as an actual argument is not eagerly evaluated but bound to the body of a new closure, named `f`, with a single parameter `x`. The variable `x` in the call-by-name argument is dynamically scoped as it is only dynamically bound to the parameter of `f` when calling `foreach`. The environment of `f` is dynamically bound upon each call of `foreach`. The newly created closure corresponds to:

```
f(x):
  if(x % 2 == 0,
    display("Even", eoln),
    display("Odd", eoln))
```

The second argument eagerly evaluated to a table and is bound to the variable, `t`.

Variadic functions. SHACL has support for variadic functions using the # notation. In Lst. 4.4 we define a new function `sum` that will add all its arguments. When `sum` is invoked, all its arguments are evaluated and stored in a single table. That table is bound to the variable `t` before the body of `sum` is evaluated. In contrast to other languages such as Scheme [Adams et al., 1998], in SHACL, there is no way to specify a minimum number of arguments for variadic functions. The function is either a fixed or indefinite arity. Similar to an `if` statement, there is no additional syntax for a `for`-loop. It is written down as a function application. Curly brackets, `{}`, are used in combination with semicolon-separated expressions as syntactic sugar for `begin` and backticks, ` `` `, are for comments.

```
sum#t:
  { result: 0;
    for(i: 1, i <= t.size(), i:= i + 1,
      result:= result + t[i]);
    result };

sum(1, 2, 3); ‘=> 6‘
```

Listing 4.4: Variable number argument lists in SHACL

Annotations. SHACL has support for annotations using the `@` notation. Currently, annotations are only used for generating scripts. See Sec. 4.2.3 for an explanation of how the `@script` annotation works in SHACL.

4.2.2. Object Oriented Programming in SHACL

The object model of SHACL was directly inspired by the object model of AmbientTalk which in turn was inspired by Prototype-Based programming languages such as Self [Ungar and Smith, 1987]. In prototype-based languages objects are not instantiated from a class but are rather created *ex-nihilo* or cloned from an existing object. In SHACL, any primitive value is an object.

Object creation. The example in Lst. 4.5 illustrates how to define a point object in SHACL. On line 1 we create a new object using the `object` primitive and bind that object to the `point` variable. There is no additional syntax for creating objects, the `object` primitive is written down as a function application with one argument, namely an expression that specifies the interface of the object. In this case, the point object has two fields, namely `x` and `y`, and two methods, `init` and `distance`.

Message sending and inheritance. Messages are used to refer to fields or invoke methods on an object. On line 10 we send a **clone** message to the `point` object to create a new point and we initialize that point by immediately sending it an `init` message. On line 12 we calculate the distance between both points by sending a `distance` message to our second point. Messages that are not understood by the object are delegated to the parent object. Each SHACL object implicitly has a special field called `super` that references the parent of the object. All primitive values and all newly created objects have the global `object` as their default parent. That field can dynamically be changed, which enables dynamic inheritance [Ungar and Smith, 1987]. The global `object` implements a number

```
point: object(
  { x: 0;
    y: 0;
    init(aX, aY):
      { x:= aX;
        y:= aY }
    distance(p):
      sqrt(sqr(p.x - x) + sqr(p.y - y)) });

other_point: point.clone().init(4, 3);

other_point.distance(point);
```

Listing 4.5: Object Definition in SHACL

of methods that allow object introspection. For example, it is possible to send the `methods` message to any object that has `object` as its parent to return all the methods of that object. The primitive `clone` is also defined as a method of `object`.

Scoping. Variables in SHACL are always looked up in the lexical environment. If the variable is not found a runtime error occurs. If a programmer want to ignore lexical scoping and qualify along which inheritance chain the variable should be looked up, he can do so by sending a message, e. g. `self.x`. In SHACL, `self` is syntax that evaluates to the current dynamic receiver.

4.2.3. Actor Oriented Programming in SHACL

The concurrent subset of SHACL is based on the model of the AmbientTalk language's communicating event-loops which in turn was based on the communicating event-loop model of E. In this model, each actor is represented as a *vat*. A vat has a single thread of execution (the event-loop), an object heap, and an event queue. Throughout this chapter, the terms *shacl actor*, *event-loop* and *vat* are used interchangeably and always refer to "actors" as described in the communicating event-loop model (See Sec. 2.1.6). A SHACL VM is started with a single "main actor". All input expressions from the REPL are evaluated as a single turn of that main actor. Other actors are created by means of the `actor` primitive. Lst. 4.6 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*. The object is initialized in the same way as the `object` primitive. 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.

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

Listing 4.6: Hello World actor in SHACL

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, the main actor 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 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. 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. 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.

Futures. SHACL includes future-type messages [Yonezawa et al., 1986] to prevent code-fragmentation that comes with the use of callbacks to retrieve the result of an asynchronous message. In SHACL, every asynchronous message is a future-type message and returns a future. That future acts as a placeholder for the return value of processing an asynchronous message. Similar to E and AmbientTalk, retrieving the result of a future is also an asynchronous operation. Retrieving the result of a future can be done using the `when_resolved(expression(value))` primitive, where `expression` is a SHACL call-by-name parameter (See Sec. 4.2.1). `expression` can be any valid SHACL expression in which `value` will be bound to the result of the future. This language mechanism is similar to SALSA's token-passing continuations (See Sec. 2.1.4) where the special keyword `token` was bound to the result of the message that was sent as part of the token-passing continuation.

```
future: a<-say_hello();

future.when_resolved(
  display("Result: ", value.to_string(), eoln));
```

Listing 4.7: Futures in SHACL

Lst. 4.7 shows an example of using future-type messages in SHACL. First, an asynchronous message `say_hello` is sent to a far reference. The result of this message is a future that is bound to the `future` variable. Retrieving the result of the message is done by registering a callback closure with the future. This is done by using the `when_resolved` primitive. This primitive expects any SHACL expression as its argument and transforms that argument into a closure with a single parameter, namely `value` and registers that closure with the future. Once the future is resolved, all registered closures are invoked with the `value` parameter bound to the result of the future. The `when_resolved` primitive is in itself an asynchronous operation. As with every asynchronous operation in

Shacl, `when_resolved` also returns a future. That future is resolved to the result of the registered closure once the callback has been executed.

```
1  make_car(parts):
2    { assembly_f: plant<-assemble(parts);
3      assembly_f.when_resolved(
4        painter<-paint(value)) };
5
6  car_f: make_car(my_parts);
7  car_f.when_resolved(
8    drive(value));
```

Listing 4.8: Chaining futures in Shacl

In Shacl, a future can also be resolved with another future, in which case the parent future is resolved with the result of the child future once the child future is resolved. Parent and child futures are *chained*. Lst. 4.8 illustrates how this is done in Shacl. On line 6, the `make_car` method is invoked to assemble a car from some car parts. Firstly, to assemble the car, on line 2, an asynchronous message is sent to the plant to assemble the parts. The resulting future, the first future, represents the promise of an assembled car and is stored in the `assembly_f` variable. A callback closure is registered with the first future using the `when_resolved` primitive. In the body of that callback, line 4, `value` is bound to the value promised by the future, in this case an assembled car. The return value of the `when_resolved` primitive is another future, the second future. That second future is also the result of the invocation of `make_car`. When the first future is resolved, the callback registered on line 3 is invoked and an asynchronous message is sent to a painter to paint the assembled parts. The result of that asynchronous message is a third future that will eventually be resolved with a painted car. The third future is also the result of the callback on line 3. This means that the second and third future are chained. Eventually the second future, the future returned from invoking `make_car` will be resolved with the painted car. On line 7, a callback is registered with the second future. Once that future is resolved, the car has been assembled and painted and can be driven. On line 8 value is bound to a painted car object.
The closure that is registered as a callback using the `when_resolved` primitive has access to its surrounding lexical scope. Because of the way actors are scoped, this lexical scope is always part of the actor that issued the callback. Thus, to avoid race conditions and to keep the isolated turn principle valid, the invocation of the callback cannot be executed in parallel with the other events of

the actor that issued the callback. In SHACL, when a future is resolved, all its registered callbacks are enqueued in the event queue of the respective actors as separate events. That means that invoking such a callback always happens in a later turn than the invocation.

```
1 future: a<-message();
2 future.when_resolved(
3   display("Then here!\n"));
4 display("First here!\n");
```

Listing 4.9: `when_resolved` is an asynchronous primitive.

For example, in Lst. 4.9 the `display` on line 4 will always be executed before the `display` on line 3. The turn of the main actor that is responsible for executing the program first needs to end before the registered callback can be processed.

Scripts. A SHACL function is always returned as a closure. Because a closure has access to its surrounding scope, a closure is always sent by *far reference* when it is passed as an argument of an asynchronous message. This severely limits the use of higher order functions across actor boundaries because a far reference to a closure cannot be called directly. To circumvent this issue SHACL introduces scripts. A *script* is a special type of SHACL function that is not returned as a closure. The body expression of a script has to be lexically closed. This implies it can only access its formal parameters and local variables, such that it is safe to pass the script by copy to another actor. The recipient actor will then receive a near reference to a copy of the script, and can call the script synchronously. To define a new script, the `@script` annotation needs to be used. Lst. 4.10 illustrates the usage of a script to find a certain account that is owned by a bank actor.

```
filter_joeri(account)@script:
  account.name == "Joeri De Koster";

bank<-find_account(filter_joeri);
```

Listing 4.10: A script in SHACL

4.3. Shared State in SHACL, A Motivating Example

In pure actor systems, shared state is often represented using a delegate actor. Sec. 3.1.2 already described a number of issues related to this approach. This section revisits these drawbacks specifically for SHACL using a small example in Lst. 4.11. This example application has three components: the bank, a client and a manager. The bank is represented by a delegate actor. The goal of the client component is to transfer money from their account to another account. The manager component can query the bank for the total sum of all account balances.

```
 1 bank: actor(
 2   { db: ‘...‘;
 3     withdraw(id, amount):
 4       db[id]:= db[id] - amount;
 5     deposit(id, amount):
 6       db[id]:= db[id] + amount;
 7     summary():
 8       sum(db.values) });
 9
10 client: actor(
11   { my_id: ‘...‘;
12     transfer(bank, other_id, amount):
13       { fut: bank<-withdraw(my_id, amount);
14         fut.when_resolved(
15           bank<-deposit(other_id, amount)) } });
16
17 manager: actor(
18   summary(bank):
19     { fut: bank<-summary();
20       fut.when_resolved(
21         display("Summary: ", value, eoln)) });
```

Listing 4.11: Motivating Example

Code fragmentation and Continuation Passing Style. Because SHACL has futures, the callback of the client actor after it has withdrawn money (line 13) can be put directly after the asynchronous message (line 14). However, because both the message send as well as the `when_resolved` primitive are asynchronous operations, using CPS is still necessary. This does not only affect the implementation of the `transfer` method. A CPS transformation will have to be applied as well to any method that calls the `transfer` method. Also, because of this transformation, the execution of the transfer method will span several turns and we lose the benefit of the isolated turn principle. The transfer method is not executed in isolation because it is possible for the manager to observe the bank in a state where the `whithdraw` method was executed but not yet the `deposit`.

No parallel reads Summing the account balances is a read-only operation. However, any two `summary` messages are always serialized by the event queue of the `bank` actor. This means that multiple managers cannot sum the account balances in parallel.
Another solution would be to query the entire database one account at a time using asynchronous messages. However, this has a number of drawbacks. Firstly, sending messages is a costlier operation that synchronously iterating over the database. Secondly, while calculating the actual sum can be done in parallel with other managers, all individual queries are still serialized by the event queue of the `bank` actor. Lastly, this solution splits the execution of the summary method over several turns, losing the benefit of the isolated turn principle. If the queries are interleaved with other `withdraw` or `deposit` messages the manager might observe a wrong total sum.
The implementation of the `summary` method was intentionally specified on the side of the bank to ensure that all read operations are done in a single turn and thus are done in isolation. However, specifying the implementation on the server side is not always feasible. For example, the manager might want to first filter some of the accounts before making the summary. It is possible to send a predicate using the `@script` annotation. However, a script is a lexically closed expression and that means that the predicate cannot depend on state that is local to the manager.
For read-only operations it is possible to send a copy of the shared state to the client actor before each operation. In this case, a copy of the database can be sent to the `manager` actor before summing the results. However, this is only beneficial if the cost of the operation is significantly greater than the cost of copying the data structure.

Message-level race conditions In the example, the `withdraw` and `deposit` messages sent by the `client` can be interleaved with the `summary` message sent by the `manager`, possibly introducing a race condition. In this case, changing the interface of the `bank` actor to include a `transfer` method would be a valid solution. However, this is not always entirely possible. Either the implementation of the bank is part of some library- or legacy code, or the messages that are sent by the `client` have to be combined in some way that is dependent on state that is local to the client. Ideally, the client would want to have synchronous access to the bank actor during the transfer method without violating the isolated turn principle.

Message-level deadlocks Because every operation in SHACL is an asynchronous operation it is impossible to provoke a deadlock. It is possible to lose progress when a user-created future is never resolved. However, futures that are the result of a message send are always resolved at some point in time. Barring any infinite loops, any SHACL turn is processed in a finite time.

4.3.1. An Idealized Implementation of the Motivating Example

In Lst. 4.11, the `bank` actor is a delegate actor that delegated any messages to the database of account values. In the abstract, the bank is not an active software entity but rather a database of values that can be accessed in a way that is specified by its interface. Ideally, rather than using a delegate actor, we would like to have an abstraction that allows us to encapsulate the state of the bank in a separate software entity that is synchronously accessible. This software entity then has to regulate any external access to the bank while maintaining the isolated turn principle.
Lst. 4.12 illustrates what an ideal application could look like. For this example, assume there exists a `shared_state` primitive that, similarly to the `actor` primitive, encapsulates all of its lexically enclosed state, the only difference being that it does not add a behavior to that state. On line 1 we instantiate the `bank` database using the `shared_state` primitive rather than the `actor` primitive. Note that `shared_state` is here purely for illustrative purposes and is not a valid SHACL primitive function. SHACL introduces a different concept to encapsulate shared state, namely a *domain*. There are various types of domains in SHACL, each with their own properties and guarantees (See Chapter 5). However, conceptually they all have the same two properties. Firstly, every domain encapsulates some state and every domain ensures that when you access that state in a particular turn, the isolated turn principle remains valid. The transfer message of the `client` actor on line 10 now synchronously withdraws and deposits money between accounts in a single turn of the client actor. The `summary` method of the `manager` actor on line 15 also synchronously sums all the account balances in a single turn of the `manager` actor. The goal of the domain abstrac-

```
bank: shared_state(
  { db: ‘...‘;
    withdraw(id, amount):
      db[id]:= db[id] - amount;
    deposit(id, amount):
      db[id]:= db[id] + amount });

client: actor(
  { my_id: ‘...‘;
    transfer(bank, other_id, amount):
      { bank.withdraw(my_id, amount);
        bank.deposit(other_id, amount) } });

manager: actor(
  summary(bank):
    display("Summary: ", sum(bank.db.values), eoln)));
```

Listing 4.12: Ideal implementation

tion is to synchronise access to the `bank` during the turn of the client and the manager in such a way that it does not violate the isolated turn principle.

4.4. Conclusion

The event-loop model has a number of desirable properties for designing component based software or plugin architectures. However, in such applications, it is common for different components or plugins to require access to a shared resource. Such a resource may be globally available for all plugins or just shared between a subset of the plugins. Currently, there is no synchronization mechanism for shared state that is tailored towards the communicating event-loop model.

5 THE DOMAIN MODEL

With the domain model we sought to introduce shared state in the communicating event-loop model by separating the event-loop and object heaps into two different concepts. In the communicating event-loop model an actor is modeled as the combination of an event-loop and an object heap. Each object in an object heap is said to be *owned* by the actor that is associated with that object heap. While an actor can own many objects, each object is owned by exactly one actor. This means that actors in this model are strictly isolated from one another and it is impossible for different actors to have direct synchronous access to the same object. In the domain model, however, object heaps are unified with a different concept called a *domain*. The main difference between a domain and a traditional object heap is that an event-loop can be associated with multiple domains and a domain is not always owned by a single event-loop. However, the various domains are still strictly isolated from one another. How an event-loop can access a domain is dependent on the type of that domain and its association with the event-loop. In this chapter we present the taxonomy that led to the design of four 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.

5.1. The Design Space: Event-loops × Object Heaps

The possible design space of combining event-loops with object heaps (i. e. domains) contains a large number of relevant points. However, starting from the notion of CELs, only a few are desirable. One axis along which we restrict the design space is by considering the number of event-loops that have synchronous access to a domain. Aiming for a programming model that provides a form of disciplined concurrency, the following property for the design space is desirable: 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:

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

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:

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

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.

5.2. Domains: Immutable, Isolated, Observable, Shared

Based on the design space resulting from combining the formulas (1) and (2), we identify five useful settings for domains. Tab. 5.1 names them in relation to the design space.

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

Tabel 5.1.: The different types of domains

Note, we will not consider *immutable isolated domains* as they are subsumed under immutable domains where the creator of the domain does not expose any reference to objects inside that domain.

Immutable domains hold data that is not owned by any particular actor and is fully immutable. This data can be read synchronously by potentially any actor.

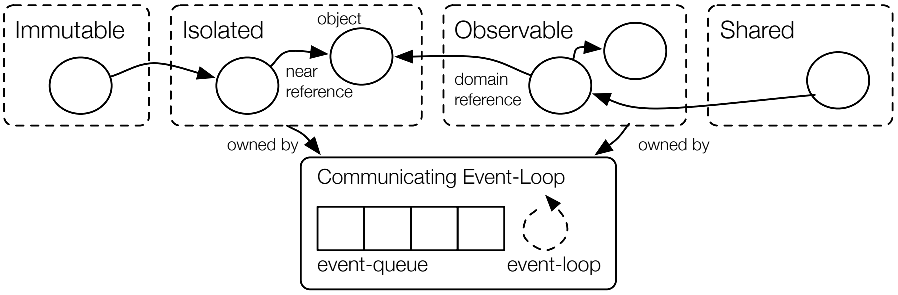
Isolated domains hold data that is owned by one actor, who has the right to read and update it synchronously. Other actors may only read or update the data in the domain asynchronously.

Observable domains hold data that is owned by one actor, who has the right to read and update it synchronously. Other actors may synchronously read the data in the domain, but can update it only asynchronously.

Shared domains hold data that is not necessarily owned by any particular actor. Any actor may synchronously read and write the data, but with an enforced multiple-reader/single-writer policy.

5.3. SHACL: A Language with Domains

In order to evaluate the usefulness of the proposed domains, the event-loop model of SHACL was extended with the notion of domains. SHACL supports all four types of domains: immutable, isolated, observable and shared domains. An instance of each of those domains is represented by the dotted boxes in Fig. 5.1. SHACL's domains are not first class, meaning that an event-loop can never hold a direct reference to a domain but rather holds references to objects inside that domain. In SHACL, a domain can be created using any of the following primitives: `immutable`, `isolated`, `observable`, and `shared`. Similar to the `actor` primitive as seen in Sec. 4.2.3, evaluating a domain primitive creates a domain of the appropriate type that is initialized with a single object for which the interface is defined by the call-by-name parameter of the domain primitive.



Figuur 5.1.: The domain model

In Fig. 5.1, 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:
The type of domain reference and the domain's relation with the event-loop determine the access capabilities of that event-loop's thread of execution to that object. To clarify, 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 model (see Chap-

ter 4). 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 `actor` primitive as seen in Sec. 4.2.3, the call-by-name parameter of each of the domain primitives has to be lexically closed. 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, *near references*, but any reference to an object belonging to a different domain is called a *domain reference*. Because of this rule, distinct domains are strictly isolated from one another.

Isolated turn principle. 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 it makes sense to enqueue that message 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.

The following section describes and evaluates for each domain type one specific design in the context of SHACL.

5.4. Immutable Domains

An immutable domain represents, as the name implies, 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.

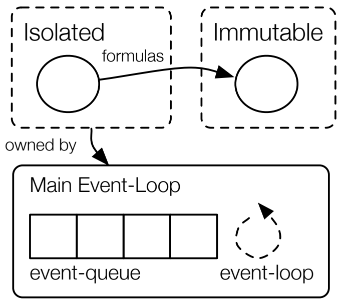
$$\forall d \in Immutable : RW(d) = \emptyset \land |R(d)| = N$$

SHACL's immutable domains are created with the `immutable` primitive. Fig. 5.2 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.

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.

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

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



Figuur 5.2.: An immutable domain defining some constants

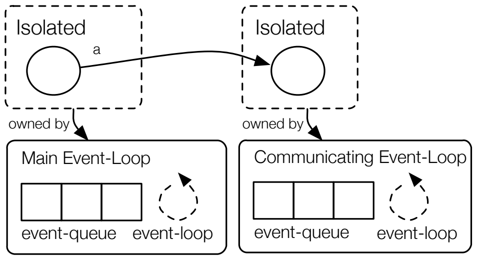
5.5. Isolated Domains

Isolated domains are most similar to the object heap of a vat in the original communicating event-loop 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 for an object heap in the original communicating event-loop model. More specifically, in SHACL, the object heap of each event-loop has been replaced with an 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.

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

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. Fig. 5.3 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();
```



Figuur 5.3.: 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 that 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 a 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.

```
o: isolated(
  say_hello():
     display("Hello World!\n"));

o.say_hello();
```

Listing 5.1: A standalone isolated domain

While not all that useful, it is possible to create a separate standalone isolated domain using the `isolated` primitive in SHACL. As shown in Lst. 5.1, the main event-loop creates an isolated domain with a single object. By creating an isolated domain, the main actor also becomes the owner of that domain and because of that it can synchronously invoke any method on any isolated domain reference pointing to objects in that domain. Creating an extra isolated domain could be useful to isolate different services within the same component from one another.

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 any 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**.

5.6. 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 SHACL, 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, if an event-loop attempts to modify a field of an observable object, that will result in a runtime error.

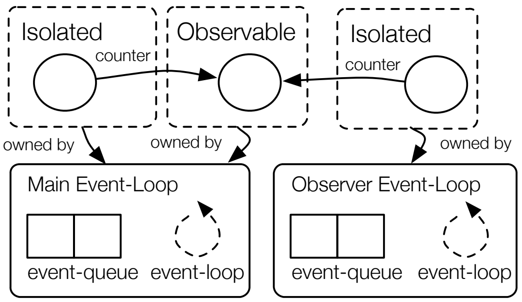
$$\forall d \in \mathit{Observable} : |RW(d)| = 1 \wedge |R(d)| = N - 1$$

Fig. 5.4 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

[1]Not to be confused with `java.util.Observable`.

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);
```



Figuur 5.4.: An observable domain owned by an event-loop

5.6.1. 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 [De Koster et al., 2013]. 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, 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.

5.6.2. Properties

The execution of the program in Fig. 5.4 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 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.

5.6.3. A Note on the Implementation

To guarantee the isolated turn principle, Shacl uses a specialized version of Software Transactional Memory [Shavit and Touitou, 1995] (See Chapter 7). 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. A 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 [Perelman et al., 2010] 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.

5.6.4. Revisiting the Motivating Example Using Observable Domains

Lst. 5.2 illustrates how we could transform the motivating example as shown in Sec. 4.3 using observable domains. The implementation of the `transfer` method has to be placed on the side of the bank because it requires a coordinated update of the bank domain. However, any read-only operations can now be executed without having to worry about coordination or consistency issues. The `summary` method on line 13 can synchronously add each of the account values. While summing the accounts, the `manager` event-loop will always observe the account database where a transfer has either happened entirely or hasn't happened yet. Also important to note is that during the turn of the `summary` message, the `manager` event-loop has synchronous access to both its own isolated domain as well as the observable bank domain.

```
bank: observable(
  { db: ‘...‘;
    transfer(id, other_id, amount):
      { db[id]:= db[id] - amount;
        db[other_id]:= db[other_id] + amount } });

client: actor(
  { my_id: ‘...‘;
    transfer(bank, other_id, amount):
      bank<-transfer(my_id, other_id, amount) });

manager: actor(
  summary(bank):
      display("Summary: ", sum(bank.db.values), eoln));
```

Listing 5.2: Transformation of our motivating example using observable domains

5.6.5. Conclusion

The idea behind observable domains is to represent state that has a single owner (and thus a single writer), but is synchronously observable by others. The owner of the domain is free to synchronously read and write to objects owned by the domain. A reference to an observable object can be freely shared between event-loops and those event-loops can **synchronously invoke** methods on those objects, the only limitation being that the invoked methods have to be read-only. From the writer's perspective the **isolated turn principle** still holds because the updates performed by the owner of the observable domain are not visible to the other event-loops for the duration of a single turn. Only at turn boundaries updates become visible for other event-loops. From the reader's perspective, the isolated turn principle also holds. An event-loop that reads from an observable domain takes the latest available snapshot of the observable domain at the start of its turn. Any consecutive reads during that turn will always yield the same values. Since no additional blocking operations are introduced and the software transactional memory used as an implementation technique does not involve any blocking operations, **deadlock freedom** is not affected either.

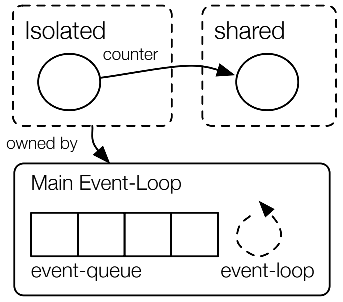
5.7. Shared domains

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));
```



Figuur 5.5.: A shared domain

Fig. 5.5 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.

5.7.1. Futures

Requesting a view using the `when_shared` or `when_exclusive` primitives are asynchronous operations. Similarly to other asynchronous operations in SHACL (e. g. asynchronous messages and the `when_resolved` primitive) they return a future. That future is resolved with the return value of the execution of the view expression associated with the request.
Lst. 5.3 illustrates the use of futures when requesting a shared view. On line 1 the main event-loop requests a shared view on the counter domain. This is an asynchronous operation that registers a view-request with the view-scheduler and immediately returns a future. In our example, that future value is stored in the `future` variable. On line 4 the main event-loop uses that future to register a closure to be executed when that future is resolved. Lastly, on line 7 the main event-loop displays a string and ends its turn. When the domain becomes available for shared access, a view is scheduled in the event queue of the main

```
future: counter.when_shared(
  display("Then here!", eoln));

future.when_resolved(
  display("Last here!", eoln));

display("First here!", eoln);
```

Listing 5.3: The when primitives return a future

event-loop. That view will only be processed by the main event-loop after it has ended its first turn (the execution of the input program text). Once the view has been processed the `future` will be resolved. When that happens, a second event, responsible for evaluating the call-by-name parameter of the `when_resolved` primitive, is scheduled in the event queue of the main event-loop.

5.7.2. Asynchronous Communication

A shared domain does not have an owner. Thus, asynchronous messages directed at a shared domain reference have to have some distinct meaning. In SHACL, an **asynchronous message** directed at a shared domain reference is syntactic sugar for requesting an exclusive view on the shared domain and in the body of the request synchronously invoking the message. Any expression of the form, `r<-m()` is translated to `r.when_exclusive(r.m())` given that `r` is a shared domain reference. Because it is impossible to statically determine if method `m` will perform any mutation, an exclusive view is taken by default regardless of whether the invocation of `m` is a read-only operation. The view that is associated with the request will eventually be enqueued in the sender's own event queue to be processed by the sender's event-loop.

5.7.3. Requesting a View on Multiple Domains

In SHACL, the idiomatic use of shared domains is to put all shared objects that have to be synchronized in the same domain. However, that also means that all updates to those objects are serialized even if those updates are to distinct objects in the shared domain (only shared views are executed in parallel). SHACL also supports a generic view primitive for when the programmer needs to synchronize access to multiple shared domains, namely the `when_acquired` primitive.

```
1  find(n, root):
2    { iterate(node, path):
3        if(node.is_void(),
4           void,
5           { path:= path + [node];
6             node.when_shared(
7               if(node.data == n,
8                  when_acquired(path, [],
9                    if(valid_path(path),
10                      node,
11                      find(n, root))),
12                 if(node.data < n,
13                    iterate(node.left, path),
14                    iterate(node.right, path)))) });
15     iterate(root, []) };
```

Listing 5.4: Using the when_acquired primitive to acquire a view on multiple domains

Lst. 5.4 illustrates how to use the `when_acquired` primitive to safely search a binary search tree where each of the nodes is in a separate shared domain. The `when_acquired` primitive takes three parameters. The first two parameters are regular call-by-value parameters which have to evaluate to two tables of shared domain references. The first table is a table of domain references on which a shared view needs to be requested. The second table is a table of shared domain references on which an exclusive view needs to be requested. When requesting a view on multiple domains SHACL uses *global lock ordering* to avoid deadlocks (See Chapter 7). When views are requested on multiple domains, all such locks should be requested together, instead of one-by-one. The third and last parameter of the `when_acquired` primitive is a call-by-name parameter that specifies the view that needs to be executed once all the domains become available for shared and/or exclusive access. In our example, each iteration of the search algorithm is executed in a different turn. That means that each step of the recursion can be interleaved with other operations on the binary search tree (e. g. insertions and deletions of other nodes). If we can guarantee that all the parent-child relations are still valid then we can guarantee that there is still a valid path between the root node and the result. On line 9 we use the `validate_path` (implementation not shown here) to check whether the followed path is still valid. However, to do this we require read access to all the nodes along the path. On line 8, we require shared access to all those nodes by using the `when_acquired` primitive. That primitive will request shared access to all the domain references in the first table and request exclusive access to all the domain references in the second table. In the example the first table contains all the nodes along the path and the second table is empty as we do not require write access to any node to validate the path. If the path is no longer valid, we restart the search (line 11). Similarly to the other when primitives, `when_acquired` is an asynchronous primitive that returns a future. While going deeper in the recursion, each future returned by `when_shared` is resolved with another `when_shared` future until we find the correct node. Once the node is found, the future of the `when_acquired` primitive is resolved with either the result or a new chain of futures. In the end, when the node is found, the whole chain of futures (See Sec. 4.2.3) is resolved with the node.

```
1  find(n, root):
2    { iterate(node):
3        if(node.is_void(),
4          void,
5          if(node.data == n,
6            node,
7            if(node.data < n,
8              iterate(node.left),
9              iterate(node.right))));
10     root.when_shared(
11       iterate(root)) };
```

Listing 5.5: Finding a node in a binary search tree

Lst. 5.5 illustrates how our algorithm would look like if all the nodes of the binary search tree were owned by the same shared domain. In this case, only a single view needs to be requested on the root node (line 10). Once the shared view has been granted we can iterate over the whole binary search tree without interference from any concurrent updates from other event-loops.

Nested view requests. In SHACL, it is considered bad practice to have dynamically nested view requests. Requesting a new view during the execution of an existing view can be problematic because, by the time the nested view is granted, the outer view will already have expired.
Lst. 5.6 illustrates this problem. On line 2, when the view on the domain of `counter` is granted, a second view is requested asynchronously. The request is scheduled and the turn of the first view immediately ends, releasing the shared view on the domain of `counter`. Once the shared domain of `other_counter`

```
1 counter.when_shared(
2   other_counter.when_shared(
3     counter.get() + other_counter.get()));
```

Listing 5.6: Nested view requests

becomes available for shared access, the second view can be processed. However, by that time the view on the domain of `counter` has already expired. The synchronous message on line 3 will result in a runtime error because the main event-loop no longer has access to the domain of `counter`.

5.7.4. Comparison with Multiple Reader/Single Writer Locks

Shared domains show a lot of similarities with multiple reader, single writer locks. However, there are a number of important differences between the two synchronization mechanisms.

```
public void increase(Counter counter,
                     ReadWriteLock lock) {
  lock.writeLock().lock();
  try {
    counter.set(counter.get() + 1);
  } finally {
    lock.writeLock().unlock();
  }
}
```

```
increase(counter):
  counter.when_exclusive(
    counter.set(counter.get() + 1));
```

Figuur 5.6.: Left, read/write lock in Java. Right, shared domain in SHACL.

Fig. 5.6 shows a comparison between an implementation of a counter using read/write locks in Java versus an implementation in SHACL using a shared domain. Both implementations first take a write lock before increasing the value of the counter.

Non-blocking.
Requesting a view is a non-blocking operation. The problem with nested locking leading to deadlocks is a known issue [Lee, 2006]. Because every potentially problematic operation in SHACL is asynchronous, the problem with deadlocks is avoided altogether. The only primitive in SHACL that acquires multiple locks simultaneously is the `when_acquired` primitive and the implementation of that primitive uses *global lock ordering* to avoid deadlocks.

Synchronization enforced.
In Java, using the read/write lock is not enforced on the client-side. Because the lock is not part of the abstract data-type, any thread is free to update the `counter` regardless of whether it holds a reference to the lock or not.

Protects whole object graph.
Any leaked references to nested objects are also protected when using a shared domain, which is typically not the case with traditional read/write locks.

5.7.5. 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 (See Chapter 7). 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`.

5.7.6. Revisiting the Motivating Example Using Shared Domains

Lst. 5.7 illustrates how we could transform the motivating example as shown in Sec. 4.3 using shared domains. This implementation is very close to the idealized implementation, as shown in Sec. 4.3.1, except for the fact that a view needs to be requested before executing the body of the `transfer` and the `summary` methods. This effectively transforms these methods into asynchronous invocations.
The multiple-reader/single-writer strategy of shared domains allows multiple `manager` actors to execute the `summary` method in parallel. But not a `summary` and a `transfer`. The `transfer` method is guaranteed to be isolated as the different accounts are owned by the same `bank` domain.
Shared domains are ideal for modeling passive software entities that do not belong to any particular actor but are shared by two or more components of the system.

```
bank: shared(
  { db: ‘...‘;
    withdraw(id, amount):
      db[id]:= db[id] - amount;
    deposit(id, amount):
      db[id]:= db[id] + amount };

client: actor(
  { my_id: ‘...‘;
    transfer(bank, other_id, amount):
      bank.when_exclusive(
        { bank.withdraw(my_id, amount);
          bank.deposit(other_id, amount) }) });

manager: actor(
  summary(bank):
    bank.when_shared(
      display("Summary: ", sum(bank.db.values), eoln)));
```

Listing 5.7: Transformation of our motivating example using shared domains

5.8. Benefits of Domains

In Chapter 3 a number of different strategies were discussed to represent shared state in modern actor systems. In the case of *pure actor systems* the more common solution was to encapsulate that shared state in a delegate actor. In Sec. 4.3, a motivating example was given to illustrate how to share state in SHACL using a delegate actor. In this chapter we have shown that domains can provide a solution for some of the issues with sharing state using a delegate actor.

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.
Any asynchronous operation always forces the programmer 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.

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

Related work on domains

Ribbons Another approach similar to our notion of domains is Hoffman et al. [2011]'s notion of ribbons to isolate state between different subcomponents of an application. They propose protection domains and ribbons as an extension to Java. Similarly to our approach, protection domains dynamically limit access to shared state from different executing threads. 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. 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 [Bocchino et al., 2009] the programmer has to use effect annotations to determine what parts (*regions*) of the heap a certain method accesses. They ensure data-race-free programs by only allowing nested calls to write disjoint sub-regions of that region. This means that this approach is best suited for algorithms that employ a divide-and-conquer strategy. In our approach we want a solution that is applicable to a wider range of problems including algorithms that randomly access data from different regions.

X10 Places In X10 [Charles et al., 2005], 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 [Armstrong et al., 1996] 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 Demsky and Lam [2010]'s views, which they propose as a coarse-grained locking mechanism for concurrent Java objects. Their approach is based on static view definitions from which, at compile time, the correct locking strategy is derived. Furthermore, their compiler detects a number of problems during compilation which can aid the developer 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 [Microsoft, 2008-09]. 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 [Baduel et al., 2006] 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 and Lain [2013] 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 [Gruber and Boyer, 2013; Haller and Odersky, 2010; Negara et al., 2011]. 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 [Scholliers et al., 2014] (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.

5.10. Conclusion

The domain model is a synchronization mechanism for shared state that is tailored towards integration with the communicating event-loop model such that the isolated turn principle holds. The isolated turn principle also holds for a number of other *pure* actor systems and the ideas discussed in this chapter can be applied in the context of those systems as well (See Chapter 9). This chapter gives a taxonomy that led to the design of 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 in a way that does not violate the properties of the original model. Each of the domain types was implemented in a communicating event-loop language called SHACL. The syntax and semantics of how to use the different domains is explained. Each domain has its separate application-dependent use-cases and some of them are shown in this chapter.

6

AN OPERATIONAL SEMANTICS FOR A SIGNIFICANT SUBSET OF SHACL

The exposition of the domain model in Chapter 5 was largely informal. In this chapter 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 [Cutsem et al., 2014] which in turn was based on that of the Cobox [Schäfer and Poetzsch-Heffter, 2010] 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.

6.1. Introduction

The full operational semantics is built in four steps. In Sec. 6.2 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 sections 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.

6.2. Basic Shacl-Lite, Actors and Their Isolated Domains

In this section 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

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

Figuur 6.1.: Semantic entities of Shacl-Lite

6.2.1. Semantic Entities

Fig. 6.1 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, do-

mains, and objects each have a distinct address or identity, denoted ι_a, ι_d and ι_o respectively.
In SHACL-LITE a **Configuration** consists of a set of live actors, A 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 ι_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 ι_o, a set of fields F, and a set of methods M. An asynchronous **Message** holds a reference r, to the object that was the target of the message, the message identifier m, and a list of values $\overline{v}$, that were passed as arguments. 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.

6.2.2. 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 ex-

Shacl Syntax

Syntax

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

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

Runtime Syntax

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

Evaluation Contexts

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

Syntactic Sugar

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

Figuur 6.2.: Syntax of Shacl-Lite

cept `this`, which means $FV(e) \subseteq \{ \texttt{this} \}$ needs to hold for all field initializer and method body expressions e. Because these expressions do not contain any free variables, actors and domains are "isolated" from their surrounding lexical scope, making them self-contained.

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 [Felleisen and Hieb, 1992] 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

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

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

Figuur 6.3.: Substitution rules: x denotes a variable name or the pseudovariable this.

6.2.3. Substitution and Tagging Rules

Substitution rules

Fig. 6.3 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

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

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

Figuur 6.4.: Runtime object tagging rules.

Tagging rules
Every object literal is tagged at runtime with the identifier of its lexically enclosing domain, ι_d, using the object_{ι_d} runtime syntax. Fig. 6.4 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.

6.2.4. Reduction Rules

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

sed 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.

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 Fig. 6.5:

- 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

$$\textsc{(let)}\quad \begin{array}{l}\mathcal{A}\langle \iota_a, Q, e_\Box[\text{let}\, x = v \,\text{in}\, e]\rangle \\ \quad \rightarrow_a \mathcal{A}\langle \iota_a, Q, e_\Box[[v/x]e]\rangle\end{array}$$

$$\textsc{(process-message)}\quad \begin{array}{l}\mathcal{A}\langle \iota_a, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \\ \quad \rightarrow_a \mathcal{A}\langle \iota_a, Q, \iota_a.\iota_o.m(\overline{v})\rangle\end{array}$$

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

$$\textsc{(field-access)}\quad \frac{\begin{array}{c} \mathcal{I}\langle \iota_a, O\rangle \in D \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f := v \in F \end{array}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_a.\iota_o.f]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[v]\rangle\}, D\rangle\end{array}}$$

$$\textsc{(field-update)}\quad \frac{\begin{array}{c} o = \mathcal{O}\langle \iota_o, F \uplus \{f := v'\}, M\rangle \\ o' = \mathcal{O}\langle \iota_o, F \cup \{f := v\}, M\rangle \\ D = D' \uplus \{\mathcal{I}\langle \iota_a, O \uplus \{o\}\rangle\} \\ D'' = D' \cup \{\mathcal{I}\langle \iota_a, O \cup \{o'\}\rangle\} \end{array}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_d.\iota_o.f := v]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\Box[v]\rangle\}, D''\rangle\end{array}}$$

$$\textsc{(congruence)}\quad \frac{a \rightarrow_a a'}{\begin{array}{c}\mathcal{K}\langle A \uplus \{a\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{a'\}, D\rangle\end{array}}$$

Figuur 6.5.: Actor-local reduction rules and congruence.

object within its associated isolated domain (with the same domain identifier, ι_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)

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

(NEW-ACTOR)

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

Figuur 6.6.: Creational rules

Rules for object and actor literals We summarize the creation reduction rules in Fig. 6.6:

- 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 ι_a reduces an actor literal expression, a new actor $\iota_{a'}$ is added to the set of actors of the configuration. A newly created isolated domain is associated with that actor. The new domain's heap consists of a single new object ι_o whose fields and methods are described by the literal expression. The $[\![e]\!]_{\iota_d}$ 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.

Asynchronous communication reductions We summarize the asynchronous communication reduction rules in Fig. 6.7:

$$\frac{}{\begin{array}{c}\mathcal{A}\langle \iota_a, Q, e_\Box[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\Box[\text{null}]\rangle\end{array}}\ (\textsc{local-asynchronous-send})$$

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

Figuur 6.7.: Asynchronous message rules

- 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`.

6.3. 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 Sec. 6.2.3. 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.

6.3.1. Semantic Entities

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

Semantic Entities of SHACL

$$\begin{array}{rcll} D \subseteq \textbf{Domain} & ::= & C \cup I & \text{Domains} \\ C \subseteq \textbf{Immutable} & ::= & \mathcal{C}\langle \iota_c, O\rangle & \text{Immutable Domains} \\ \multicolumn{4}{r}{\iota_c \in \textbf{ImmutableId}, \textbf{ImmutableId} \uplus \textbf{IsolatedId} \subseteq \textbf{DomainId}} \end{array}$$

Figuur 6.8.: Additional semantic entities for immutable domains

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

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

Figuur 6.9.: Additional syntax for immutable domains

6.3.2. 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)

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

(IMMUTABLE-FIELD-ACCESS)

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

Figuur 6.10.: Immutable Domain Actor-Local Reduction Rules.

6.3.3. Reduction Rules

Actor-local reductions We summarize the actor-local reduction rules in Fig. 6.10:

- 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)

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

(NEW-IMMUTABLE-DOMAIN)

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

Figuur 6.11.: Immutable Creational Rules

Rules for object and immutable domain literals We summarize the immutable domain creation reduction rules in Fig. 6.11:

- 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.

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

Figuur 6.12.: Immutable Asynchronous Message Reduction Rules

Asynchronous communication reductions We summarize the asynchronous communication reduction rule in Fig. 6.12:

- 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`.

6.4. 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

$$
\begin{array}{rcll}
D \subseteq \textbf{Domain} & ::= & C \cup I \cup B & \text{Domains} \\
B \subseteq \textbf{Observable} & ::= & \mathcal{B}\langle \iota_b, \iota_a, f, O \rangle & \text{Observable Domains} \\
\multicolumn{4}{l}{\iota_b \in \textbf{ObservableId}, \textbf{IsolatedId} \,\dot\cup\, \textbf{ImmutableId} \,\dot\cup\, \textbf{ObservableId} \subseteq \textbf{DomainId}}
\end{array}
$$

Figuur 6.13.: Additional semantic entities for observable domains

6.4.1. Semantic Entities

The set of domains is extended with the set of observable domains. An **Observable** domain has an identifier (ι_b) and an identifier that specifies the owner of the domain (ι_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**.

6.4.2. 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.

6.4.3. Reduction Rules

Actor-local reductions We summarize the actor-local reduction rules in Fig. 6.15:

SHACL Syntax

Syntax

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

Runtime Syntax

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

Evaluation Contexts

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

Figuur 6.14.: Additional syntax for observable domains

- PROCESS-MESSAGE: this rule replaces the rule for processing messages in Fig. 6.5. 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, ι_a. Any actor can invoke a method on an observable domain reference, regardless of the owner of the domain. A method invocation reduces the

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

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

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

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

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

Figuur 6.15.: Observable Domain Actor-Local Reduction Rules.

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.

Rules for object and observable domain literals We summarize the immutable domain creation reduction rules in Fig. 6.16:

(NEW-OBSERVABLE-OBJECT)

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

(NEW-OBSERVABLE-DOMAIN)

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

Figuur 6.16.: Observable Creational Rules

- 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, ι_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.

Asynchronous communication reductions We summarize the asynchronous communication reduction rules in Fig. 6.17:

- 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

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

Figuur 6.17.: Observable Asynchronous Message Reduction Rules

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

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

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,

ι_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.

6.5. 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**	$::=$	$\text{SH} \mid \text{EX}$	Request Types
$l \in$ **AccessModifier**	$::=$	$\text{R}(n) \mid \text{W} \mid \text{F}$	Access Modifiers
$\iota_a \in S, E \subseteq$ **IsolatedId**			Actor Identifiers
$i \in \mathbb{N}$			Integers

Figuur 6.18.: Additional semantic entities for shared domains

6.5.1. Semantic Entities

The set of domains is extended with the set of shared domains. A **Shared** domain has an identifier, ι_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 ι_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**.

SHACL Syntax

Syntax

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

Runtime Syntax

$$e \quad ::= \quad \ldots \mid \text{release}_\text{t}(\text{v})$$

Evaluation Contexts

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

Syntactic Sugar

$$\begin{aligned} \text{whenShared}(e)\{e'\} &\overset{\text{def}}{=} \text{acquire}_{\text{SH}}(e)\{e'\} \\ \text{whenExclusive}(e)\{e'\} &\overset{\text{def}}{=} \text{acquire}_{\text{EX}}(e)\{e'\} \end{aligned}$$

Figuur 6.19.: Additional syntax for shared domains

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

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

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.

$$\textsc{(invoke)}\quad \frac{r = \iota_s.\iota_o \qquad \iota_a \in S \cup E \qquad \mathcal{S}\langle \iota_s, l, S, E, R, O\rangle \in D \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \mathrm{m}(\overline{x})\{e\} \in M}{\mathcal{K}\langle A \cup\!\!\!\cdot\, \{\mathcal{A}\langle \iota_a, Q, e_\square[r.m(\overline{v})]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\}, D\rangle}$$

$$\textsc{(field-access)}\quad \frac{\iota_a \in S \cup E \qquad \mathcal{S}\langle \iota_s, l, S, E, R, O\rangle \in D \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f := v \in F}{\mathcal{K}\langle A \cup\!\!\!\cdot\, \{\mathcal{A}\langle \iota_a, Q, e_\square[\iota_s.\iota_o.f]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[v]\rangle\}, D\rangle}$$

$$\textsc{(field-update)}\quad \frac{\iota_a \in E \\ o = \mathcal{O}\langle \iota_o, F \cup\!\!\!\cdot\, \{f := v'\}, M\rangle \\ o' = \mathcal{O}\langle \iota_o, F \cup \{f := v\}, M\rangle \\ D = D' \cup\!\!\!\cdot\, \{\mathcal{S}\langle \iota_s, \mathrm{w}, S, E, R, O \cup\!\!\!\cdot\, \{o\}\rangle\} \\ D'' = D' \cup \{\mathcal{S}\langle \iota_s, \mathrm{w}, S, E, R, O \cup \{o'\}\rangle\}}{\mathcal{K}\langle A \cup\!\!\!\cdot\, \{\mathcal{A}\langle \iota_a, Q, e_\square[\iota_s.\iota_o.f := v]\rangle\}, D\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, Q, e_\square[v]\rangle\}, D''\rangle}$$

Figuur 6.20.: Shared Domain Actor-Local Reduction Rules.

6.5.3. Reduction Rules

Actor-local reductions We now summarize the actor-local reduction rules in Figure 6.20:

- 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 ($\iota_a \in E$).

(NEW-OBJECT)

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

(NEW-SHARED-DOMAIN)

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

Figuur 6.21.: Shared Creational Rules

Rules for object, domain and actor literals We summarize the creation reduction rules in Figure 6.21:

- 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 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.

(SHARED-ASYNCHRONOUS-SEND)

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

Figuur 6.22.: Shared Asynchronous Message Reduction Rules

Asynchronous communication reductions We summarize the asynchronous communication reduction rules in Figure 6.22:

- 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`.

(ACQUIRE-VIEW)

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

(PROCESS-VIEW-REQUEST)

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

(PROCESS-VIEW-NOTIFICATION)

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

(RELEASE-VIEW)

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

Figuur 6.23.: Views reduction rules.

View reductions We summarize the view reduction rules in Figure 6.23:

- 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, ι_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 Sec. 6.4).

- 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

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

6.6. Differences Between Shacl and Shacl-Lite

The operational semantics for Shacl-Lite was based on the semantics for the AmbientTalk language [Cutsem et al., 2014]. 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.

While the operational semantics gives a correct formal specification of the semantics of the different domains, it certainly does not reflect an optimal implementation strategy. A possible implementation of the domain model is discussed in Chapter 7.

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.

6.7. 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.

7 IMPLEMENTATION: DOMAIN HANDLERS

The implementation of the SHACL virtual machine is an abstract syntax tree interpreter. The source code for any SHACL program is translated to an AST by the reader and evaluated by the evaluator. That evaluator is a reentrant implementation that is parametrized by a runtime context. That runtime context stores, among other things, the lexical domain in which the current expression is being evaluated. In the implementation of SHACL each domain is linked to a specific *domain handler*. Each domain handler implements a number of so-called "traps", methods that can be triggered by the evaluator. When a relevant operation is executed, the evaluator calls the corresponding trap. In other words, domain handlers are the internal API with which the different domains are implemented. This is a generic API that allows specification of the different domains by means of redefining the different traps. In this chapter we focus on this aspect of the implementation by presenting each individual domain handler for the four types of domains.

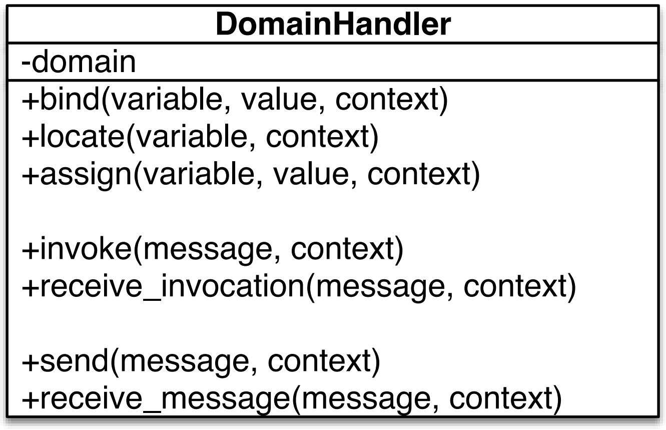
7.1. The Shacl VM: An Implementation in Go

The Shacl Virtual Machine was implemented in the Go Programming Language [Google, 2011]. The Go programming language is a systems language that is recognizably in the tradition of C but has some features that make it more accessible. The most important ones in the context of Shacl are: a garbage collector and a lightweight concurrency mechanism in the form of go-routines. The original idea was that the combination of a garbage collector with lightweight concurrency would allow us to experiment with scalable parallelism without having to worry about those extra concerns. The Go implementation of the Shacl VM was implemented as an abstract syntax tree interpreter with a read-eval-print loop. The current implementation has about 5600 lines of Go code that can be found online [De Koster, 2014]. This implementation serves as a platform for experimenting with new language abstractions for the communicating event-loop model. There are many advantages of designing a new programming language from scratch over a implementing a library for an existing language. Most importantly, a programming language allows us to enforce certain properties of our model. In Chapter 8, we will discuss an implementation of Domains in Scala as a library, where we will highlight the limitations of the library approach. Rather than focussing on optimization, the implementation of Shacl focusses on reproducing correct semantics, and providing lightweight syntax for the domain model.

7.2. The Domain Handler Interface

In the current implementation of Shacl each expression is evaluated with respect to a certain *runtime context*. A runtime context is defined by the current **dynamic receiver**, the **lexical environment**, the **lexical domain** and the currently executing **actor**. The current receiver is a pointer to the late-bound receiver *self*. The environment represents the current lexical scope, i.e. a map from visible variables to their values. The lookup of receiver-less messages will start in the environment. Every lexical environment is always part of its lexically enclosing domain and is subject to that domain's restrictions. For example, an immutable domain enforces immutability of all of its lexically enclosed environments. Because the lexical environment does not have an explicit back-pointer to its lexically enclosing domain, the lexical domain is also part of the context parameter.

Fig. 7.1 shows the interface of a domain handler. The *domain* is the domain that is associated with the domain handler. Each domain handler can define up to seven different traps. If a particular trap is not defined the default one is used.


DomainHandler
-domain
+bind(variable, value, context)
+locate(variable, context)
+assign(variable, value, context)
+invoke(message, context)
+receive_invocation(message, context)
+send(message, context)
+receive_message(message, context)

Figuur 7.1.: UML class diagram of the domain handler interface

The different traps are introduced below. In the signature of the different traps the `context` parameter is the context from which the trap was called. `variable` should always be a symbol while `value` can be any SHACL runtime value. A `message` holds the receiver object, the method name as a symbol and a list of arguments. One important observation to make is that in SHACL object fields and variables are unified. The `bind`, `locate` and `assign` traps apply for both variables and object fields.

`bind(variable, value, context)` defines a new variable bound to a value in the current lexical environment. The lexical environment is always grouped with its lexically enclosing domain in the context. This means that the bind trap is always invoked on the domain handler of the lexically enclosing domain.

`locate(variable, context)` looks up the value that is bound to the variable in the current lexical environment. Similarly to `bind`, this trap is always invoked on the lexically enclosing domain.

`assign(variable, value, context)` rebinds the variable to a new value in the current lexical environment. Similarly to `bind` and `locate`, this trap is also always invoked on the lexically enclosing domain.

`invoke(message, context)` is triggered on outgoing synchronous messages. Each time a method is invoked, this trap is called on the domain handler of the current lexical domain.

`receive_invocation(message, context)` is triggered on incoming method invocations. The receiver object of the invocation can potentially be owned by a different domain than the one from which the invocation was made.

`send(message, context)` is triggered on outgoing asynchronous messages. Similarly to `invoke`, this trap is called on the domain handler of the current lexical domain.

`receive_message(message, context)` is triggered on incoming asynchronous messages. Similarly to `receive_invocation`, if the receiver of the asynchronous message is a domain reference, this can potentially be called on a different domain handler.

Domain handlers are currently not first class entities in SHACL. They are implemented in Go and the implementation directly calls into them from the evaluator. However, in view of giving a concise description of the implementation of the different domain handlers, this chapter shows a possible meta-circular implementation written in SHACL. Firstly the default domain handler is presented. The default domain handler defines the default behavior for each of the traps. Other domain handlers can specialize this implementation by overriding the different traps. Afterwards the implementation for the domain handlers of the four types of domains is shown.

7.2.1. Default Domain Handler

The default domain handler defines the default behavior for each of the traps. Lst. 7.1 shows a possible implementation of the default domain handler.

`domain`. There is a bidirectional association between a domain and its domain handler.

`bind`, `locate`, `assign`. These are the three traps that are triggered upon defining, referring to and assigning a variable or field. The default trap delegates the call to the current lexical environment which then adds, looks up or assigns the variable in the appropriate scope.

`invoke`. The default implementation for method invocation first checks the type of the receiver object. If the receiver is a domain reference then the actual method invocation will be handled by the domain of the receiver object. On line 19 we unwrap the receiver object. The message is then further processed by invoking the `receive_invocation` trap on the domain handler of the owner of the unwrapped object. If the receiver object is not a domain reference the processing of the invocation is simply forwarded to the `receive_invocation` trap of the same domain (i. e. `self`).

`receive_invocation`. On line 25 the context in which the method will be executed is switched to the domain of the domain handler. The domain in which the method is invoked is always the domain that owns the receiver object.

`send`. Similarly to method invocation, processing an asynchronous message also forwards the actual processing of the message to another trap of the domain handler of the domain that owns the receiver object, in this case the `receive_message` trap.

`receive_message`. An asynchronous message in SHACL is always translated to a synchronous method invocation that is enqueued as an event in an event queue. The default implementation will enqueue that event in the sender's own event queue and return the corresponding future as the result of the asynchronous message. When the message is processed the corresponding method is invoked on the target object and the future is resolved with the return value of that invocation.

```
default_domain_handler: object(
  { domain: void;

    initialize(a_domain):
      domain:= a_domain;

    bind(variable, value, context):
      context.environment.bind(variable, value);

    locate(variable, context):
      context.environment.locate(variable);

    assign(variable, value, context):
      context.environment.assign(variable, value);

    invoke(message, context):
      { receiver: message.receiver;
        if(receiver.type == "domain_reference",
           { message.receiver := receiver.dereference();
             receiver.domain.handler.receive_invocation(message, context) },
           self.receive_invocation(message, context)) };

    receive_invocation(message, context):
      { ctx: context.clone();
        ctx.domain := domain;
        message.invoke(ctx) };

    send(message, context):
      { receiver: message.receiver;
        if(receiver.type == "domain_reference",
           { message.receiver := receiver.dereference();
             receiver.domain.handler.receive_message(message, context) },
           self.receive_message(message, context)) };

    receive_message(message, context):
      { event: create_event(message, context);
        context.actor.event_queue.enqueue(event);
        event.future } });
```

Listing 7.1: Default Domain Handler

7.2.2. Immutable Domain Handler

An immutable domain is an object heap of immutable objects that are freely accessible by all the different actors in the system. In that sense the immutable domain handler mostly corresponds to the default domain handler. Actors are free to define and lookup variables and to invoke methods on immutable domain references. Remember from Sec. 5.3 that asynchronous messages that are sent to domain references of domains without an owner are enqueued in the sender's own event queue. Equally, asynchronous messages that are sent to an immutable domain reference are enqueued in the actor's own event queue. The only entry point to an immutable domain is through method invocation. Invoking a method will lead to the creation of a new scope and any variables that are declared during that method invocation are bound in that new scope. Because that new scope is local to the actor executing the invocation, two actors can never define a new variable in the same scope. This means that defining new variables in an immutable domain can be allowed without the risk of possible race conditions. The only trap that is overridden is `assign` to prevent actors from mutating the heap of an immutable domain.
Lst. 7.2 illustrates a possible meta-circular implementation. The immutable domain handler inherits from the default domain handler by making a clone of that handler. Attempting to assign to a variable of an immutable domain will result in a runtime error.

```
immutable_domain_handler: object(
  { super:= default_domain_handler.clone();

    assign(variable, value, context):
      error("Not allowed to write to immutable domain.") });
```

Listing 7.2: Immutable Domain Handler

7.2.3. Isolated Domain Handler

An isolated domain is always associated with a single actor, namely the owner of that domain. Only the owner of an isolated domain can read and write to objects owned by that domain. Similar to Smalltalk [Goldberg and Robson, 1983], in SHACL, object fields are private and methods are public. Accessing a field of an object is translated to a method invocation. `object.field` is translated to `object.field()` and `object.field := expression` is translated to `object.field!(expression)`. This means that method invocations are the only entry points to a domain. In the default implementation shown in Lst. 7.1 the `receive_invocation` is the one responsible for switching the domain in the context. To ensure isolation for isolated domains it is sufficient to check whether the actor that is accessing the domain is the owner of the domain. Lst. 7.3 gives a possible meta-circular implementation for an isolated domain handler. The default domain handler is extended with an `owner` field and the `receive_invocation` and `receive_message` traps are overridden. The `receive_invocation` trap first checks whether the actor that is currently executing the invocation is also the owner of the isolated domain before delegating the method invocation to the default domain handler. If the currently executing actor is not the owner then a runtime error is thrown. Asynchronous messages sent to an isolated domain reference are enqueued in the event queue of the owner of the isolated domain rather than in the sender's own event queue. The `receive_message` is overridden to implement this semantics.

```
isolated_domain_handler: object(
  { super:= default_domain_handler.clone();
    owner: void;

    initialize(a_domain, an_actor):
      { owner:= an_actor;
        super.initialize(a_domain) };

    receive_invocation(message, context):
      if(context.actor == owner,
         super.receive_invocation(message, context),
         error("Not allowed to synchronously access isolated domain"));

    receive_message(message, context):
      { event: create_event(message, context);
        owner.event_queue.enqueue(event);
        event.future } });
```

Listing 7.3: Isolated Domain Handler

7.2.4. Observable Domain Handler

Observable domains are domains that are owned by a single actor but, in contrast to isolated domains, are observable by others. Similarly to isolated domains, the owner of an observable domain is allowed to synchronously read and write to objects owned by that domain. However, other actors can also synchronously read from, but not write to, objects owned by that domain. To ensure the isolated turn principle, during any turn, the observers of the domain should always observe an immutable consistent snapshot of the domain. In later turns an observer might observe a newer version of the domain but during a single turn the domain has to appear to be immutable. To guarantee that other actors always see the same version of an observable domain, SHACL uses a Multi-Version History Software Transactional Memory [Perelman et al., 2010] system. In SHACL each turn executed by an actor is considered to be a transaction. However, it is important to note that while in SHACL a turn has a transactional behavior, the domain model does not have any STM specific keywords to delimit transactions. To support the multi-version history STM, SHACL actors were extended with an `age` field which specifies the age of an actor in terms of the number of turns it has successfully completed since its creation. For observable domains this means that the age in turns of the owner of an observable domain (`owner.age`) determines the version number of the last committed version of an observable domain. Additionally, actors are also extended with an associative data-structure, the `version_store`, to specify what version of a domain an actor is currently reading. The observable domains owned by an actor all share the same version, namely its age. The `version_store` relates to versions of the domains owned by other actors. At the start of any turn, the `version_store` is empty and it is only updated when an actor first attempts to access an observable domain. This means the cost in terms of performance is only paid when observable domains are actually used.

```
1  event_loop():
2    { version_store.empty();
3      event: event_queue.dequeue();
4      process_event(event);
5      age:= age + 1;
6      event_loop() }
```

Listing 7.4: The event-loop of a SHACL actor

Lst. 7.4 illustrates a possible meta-circular implementation of the event-loop of a SHACL actor. On line 2, before starting the turn the actor empties the store of versions. During the following turn the actor will read from the latest consistent snapshot of any observable domain it accesses. Afterwards the actor dequeues and processes the next event from its queue. During the processing of this event the `version_store` will be populated with the version that is being read from observable domains for which the actor is not the owner. On line 5, after the event is processed we have successfully finished the turn and the actor can increase its age.

Lst. 7.5 gives a possible meta-circular implementation for an observable domain handler. The software transactional memory is implemented by means of three functions not shown here, namely `stm_bind(variable, value, version, environment)`, `stm_locate(variable, version, environment)` and `stm_assign(variable, value, version, environment)`. `stm_bind` creates a new binding in the environment and sets up the STM for that binding. `stm_assign` locates the `variable` in the `environment` and appends a new version-value pair to the existing set of versions for that variable. If the version already exists, the value in the version-value pair is overwritten. `stm_locate` locates the variable in the environment and finds the most recent version-value pair for which the version number is lower than the given version number. That means the returned value is always as old or older than the given version number.

The `locate` trap is shown on line 11. If the currently executing actor is the owner of the domain we always read the latest uncommitted version (`owner.age + 1`). Otherwise, on line 16 the version that is currently being read by the actor is looked up. If this is the first time the current actor is reading from the domain then we register the latest committed version (`owner.age`) on line 19. Until the end of its current turn, read operations of the current actor on the same domain will yield the same result. This is because the version read by an actor is stored in the `version_store` per domain and is only emptied at the start of the next turn.

Similar to immutable domains, actors with read-only access to an observable domain can create new variable-value bindings. Invoking a method on an observable domain reference will lead to the creation of a new scope that is local to the actor and any variables that are declared during that method invocation are bound in that new scope. The implementation of the `bind` trap is not shown but is similar to the implementation of `locate`. If the currently executing actor is the owner of the domain the new binding is initialized, using `stm_bind`, with a new version-value pair with the latest uncommitted version (`owner.age + 1`) as its version number. Otherwise, the binding is created with the version number that is currently being read by the the actor (`context.actor.get_version(self.domain)`).

```
observable_domain_handler: object(
  { super:= default_domain_handler.clone();
    owner: void;

    initialize(a_domain, an_actor):
      { owner:= an_actor;
        super.initialize(a_domain) };

    bind(variable, value, context): ‘...‘;

    locate(variable, context):
      if(context.actor == owner,
         stm_locate(variable,
                    owner.age + 1,
                    context.environment),
         { version: context.actor.get_version(self.domain);
           if(not(version),
              { version:= owner.age;
                context.actor.set_version(self.domain, version) });
           stm_locate(variable,
                      version,
                      context.environment) });

    assign(variable, value, context):
      if(context.actor == owner,
         stm_assign(variable,
                  value,
                    owner.age + 1,
                    context.environment),
         error("Only the owner of a domain can assign a new value"));

    receive_message(message, context):
      { event: create_event(message, context);
        owner.event_queue.enqueue(event);
        event.future } });
```

Listing 7.5: Observable Domain Handler

The `assign` trap is shown on line 24. If the currently executing actor is the owner of the domain then a new version-value pair is appended to the list of versions for the `variable` in the `environment`. Unless the same variable was written to before on that same turn, the version-value pair is simply overwritten. A write is always done to the latest uncommitted version (`owner.age + 1`). If the currently executing actor is not the owner of the domain a runtime error is thrown.
Similar to isolated domains, the `receive_message` trap is also overridden to enqueue incoming messages in the event queue of the owner of the domain rather than the sender's own event queue.

7.2.5. Shared Domain Handler

A shared domain does not have a particular owner. Any actor can have synchronous read and write access to objects in a shared domain by first acquiring a view. Lst. 7.6 gives a possible meta-circular implementation for an shared domain handler. The default domain handler is extended with a set of `requests`, a list of actors that currently have read access to the domain, `readers`, and a potential `writer` that has read and write access to the domain. The `readers` correspond to the actors that currently have a shared view on the domain while the writer corresponds to an actor that has an exclusive view on the domain. The `status` field represents the lock of the shared domain. The `status` is `0` when the domain is free for shared or exclusive access, the `status` is `-1` when the domain is locked for exclusive access and the `status` is any non-zero integer when the domain is locked for shared access. Similar to isolated domains, because fields are private in SHACL, the only entry point for actors into a shared domain is through invoking a method on a shared domain reference. Invoking a method on a shared domain reference can be done by any actor that currently has a shared or exclusive view on that domain. On line 14, the `receive_invocation` trap checks whether the currently executing actor has read access to the domain. If this is not the case a runtime error is thrown. During the invocation of the method any `bind` or `locate` operations are valid. However, assigning a new value to a variable can only be done from within an exclusive view. This is checked by the `assign` trap. If the currently executing actor does not currently have an exclusive view on the domain, a runtime error is thrown.
The SHACL `when_shared` and `when_exclusive` built-in functions are translated to calls to `add_shared_request` and `add_exclusive_request` respectively. Adding a request groups the body expression of the built-in function with the current context. A future value is also associated with that request and is imme-

diately returned to the actor that issued the request. Once the domain becomes available for shared or exclusive access, the body expression is enqueued as an event in the event queue of that actor. Once that event is processed by the actor, the future is resolved with the return value of calling that closure.
The `receive_message` trap translates asynchronous messages directed at a shared domain reference into an exclusive request. The body expression of the exclusive request is a synchronous invocation of the same method on the shared domain reference. The message is translated into an exclusive request because the interpreter does not now statically whether the method invocation is read-only or not.

```
shared_domain_handler: object(
  { super:= default_domain_handler.clone();
    requests: [];
    readers: [];
    writer: void;
    status: 0;

    assign(variable, value, context):
      if(context.actor == writer,
         context.environment.assign(variable, value),
         error("Can only write to a shared domain from an exclusive view"));

    receive_invocation(message, context):
      if(or(context.actor == writer, readers.include?(context.actor)),
         super.receive_invocation(message, context),
         error("Can only invoke methods on a shared domain from a view"));

    receive_message(message, context):
      add_exclusive_request(message, context);

    add_exclusive_request(expression, context):
      { future: requests.add_exclusive(expression, context);
        handle_requests();
        future };

    add_shared_request(expression, context):
      { future: requests.add_shared(expression, context);
        handle_requests();
        future } });
```

Listing 7.6: Shared Domain Handler

Lst. 7.7 shows how shared and exclusive requests are handled in the implementation. Every time a request is added or every time a view has completed, other requests for views on the same domain are processed by invoking `handle_requests`. To handle the next request, on line 2, we first attempt to get an arbitrary exclusive request from the set of requests. If an exclusive request was found and there are currently no readers and no writers of the domain (i. e. `status == 0`) then that request can be granted. The status field is set to `-1` to denote that the shared domain has granted exclusive access to an actor. On line 6 an exclusive view is created and is enqueued in the event queue of the actor that originally requested the view. At this time, the `writer` field of the shared domain is still `void`. That field is only changed when the view-event is processed by the actor, thereby granting exclusive access to the shared domain to that actor (See Lst. 7.8). If the `status` $\neq$ `0` and there is an exclusive request available no other requests are handled to prevent starvation of the exclusive requests. If there is currently no exclusive request granted (`status >= 0`) then all pending shared requests can be handled. For each request that is handled, the `status` field is increased by one. On line 16 we create a new view for each pending shared request and enqueue those views in the event queue of the corresponding actors.

```
1  handle_requests():
2    { exclusive_request: requests.get_exclusive_request();
3      if(exclusive_request,
4         if(status == 0,
5            { status:= -1;
6              view: create_exclusive_view(exclusive_request.expression,
7                                          exclusive_request.future,
8                                          exclusive_request.context);
9              exclusive_request.context.actor.event_queue.enqueue(view) },
10           void), ‘stop here to prioritize writers‘
11        { shared_requests: requests.get_shared_requests();
12          if(and(shared_requests, status >= 0),
13             shared_requests.each(
14               { shared_request: element;
15                 status:= status + 1;
16                 view: create_shared_view(shared_request.expression,
17                                          shared_request.future,
18                                          shared_request.context);
19                 request.context.actor.event_queue.enqueue(view) })) }) };
```

Listing 7.7: Handling Requests

Lst. 7.8 shows how the different views are processed. When a view is granted and put into the event queue of the actor that requested the view, the actor does not yet have access to the shared domain. Only when the view event is processed by that actor, is shared or exclusive access granted. For example, when processing an exclusive request on line 3, the actor first adds himself as the `writer` before evaluating the body expression of the view. The future that was associated with that view request is resolved with the result of that evaluation. Once the body expression is evaluated, the view can be released by nullifying the `writer` field and setting the `status` field to `0`. Afterwards, any other pending requests can be handled. Processing a shared view works in a similar way. On line 11 the actor adds itself as a reader of the domain. After the body expression has been evaluated, the actor removes itself from the set of `readers` and decrements the `status` field before handling any other pending requests.

```
1  process_exclusive_view(view):
2    { handler: view.context.domain.handler;
3      handler.writer := self;
4      view.future.resolve(evaluate(view.expression, view.context));
5      handler.writer := void;
6      handler.status := 0;
7      handler.handle_request() }
8
9  process_shared_view(view):
10   { handler: view.context.domain.handler;
11     handler.readers.add(self);
12     view.future.resolve(evaluate(view.expression, view.context));
13     handler.readers.remove(self);
14     handler.status := handler.status - 1;
15     handler.handle_request() }
```

Listing 7.8: Processing a View

7.3. Conclusion

This chapter presents *domain handlers* as a generic way to specify the semantics of the different domains in SHACL. The traps allow the domain handlers to specify custom behavior for defining, referencing and modifying variables. A domain handler can also specify custom behavior for incoming and outgoing synchronous and asynchronous messages. This chapter presents a meta-circular implementation of the different domain handlers used in SHACL for immutable, isolated, observable and shared domains. However, because of the generic specification of domain handlers this technique could be used to extend SHACL with various other domains.

8 APPLYING DOMAINS IN PRACTICE: A CASE STUDY IN SCALA

In an impure actor system such as the Scala or Akka actor library, programmers already have a number of alternatives when it comes to representing shared state. On the one hand, they can fall back on the underlying shared-memory model and use traditional mechanisms such as locks to synchronize access to a shared resource. On the other hand, they can employ the actor model for synchronization by encapsulating a shared resource in a delegate actor. This chapter shows that these two synchronization mechanisms are being used in practice by executing a survey of a relevant set of existing Scala projects. In addition, to further motivate why programmers might prefer one synchronization mechanism over the other, the advantages and disadvantages of each approach are discussed. Afterwards a possible implementation of domains for Scala is shown and the domain model is validated by providing an alternative implementation for some of the code-examples found in the survey. The chapter is concluded by showing that the domain model has a number of desirable properties over the other synchronization mechanisms found in the survey.

8.1. Shared State Synchronization Patterns: A Scala Survey

A recent study [Tasharofi et al., 2013] on 16 large, mature, and actively maintained actor programs written in Scala has found that 80% of them mix the actor model with another concurrency model for synchronizing shared state. When asked for the reason behind this design decision, one of the main motivations programmers brought forward were some of the inadequacies of the actor model when it comes to shared state, stating that certain protocols are easier to implement using shared-memory than using asynchronous communication mechanisms without shared state. In Scala, developers can fall back on the underlying shared-memory concurrency model for modeling access to a shared resource.

8.1.1. The Corpus of Actor Programs

For our case study we reuse the corpus of [Tasharofi et al., 2013] which is publicly available online.[1] From the initial set of around 750 Scala programs available on github,[2] 16 real-world actor projects were selected based on the following criteria:

- **Actor Library**: The program uses the Scala or Akka actor library to implement a portion of its functionality.
- **Size**: The program must consist of at least 3000 lines of code combined over Scala and Java.
- **Eco-System**: At least two developers contribute to the project.

For our case study we updated the corpus to the latest version available on github. Tab. 8.1 gives an overview of the corpus used in the survey. The **Project** column shows the project's name on github. The **Library** column indicates which actor library was used in the project. The lines of code (**LOC**) were counted using the CLOC tool.[3] This tool allows us to automatically detect which language was used in each file and separates newlines and comments from the actual lines of code. The **Description** column gives a short description of the application.

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

Project	Library	ScalaLOC	JavaLOC	Description
bigbluebutton	Scala	9459	65855	Web conferencing system
blueeyes	Akka	23107	0	A web framework for Scala
CIMTool	Scala	3915	26562	A modeling tool based on Common Information Model (CIM) standards
diffa	Akka	29947	5985	Real-time data differencing
ensime	Scala	8765	43	Enhanced Scala Interaction Mode for Emacs
evactor	Akka	4743	0	Complex event processing
gatling	Akka	19759	105	A stress test tool
geotrellis	Akka	54200	7778	Geographic data engine
kevoree	Scala	9694	37325	A tool for modeling distributed systems
SCADS	Scala	27119	963	Distributed storage system
scalatron	Akka	12248	0	A multi-player programming game
signal-collect	Akka	12635	0	Parallel graph processing framework
socko	Akka	12505	121	A Scala web server
spark	Scala	116983	7648	Cluster computing system
spray	Akka	30770	0	RESTful web services library
ThingML	Scala	10950	64238	Modeling language for distributed systems

Tabel 8.1.: The corpus of projects used in the survey

8.1.2. Evaluation of The Different Synchronization Mechanisms

This section gives an overview of a number of desirable properties for any synchronization mechanism. The goal of this chapter is to evaluate the different synchronization patterns found in the survey according to these properties. Throughout this chapter we will refer to the source-code that models the shared resource as the *server-side* code while the source-code that models the access to that shared resource will be referred to as the *client-side* code.

- **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.
- **Deadlock free**. Blocking synchronization mechanisms do not suffer from this 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.

8.1.3. The Survey: Locks and Delegate Actors

For our case study we chose to investigate the use of two patterns to synchronize access to shared state, namely locks and delegate actors. Those two synchronization mechanisms are the two most prevalent patterns found in the corpus. We opted to do a syntactic source code analysis. While that lowers the precision over more advanced tools such as byte-code analyzers, we do feel confident about the results as all actor classes are easily identified by finding classes that implement an `act` (Scala) or `receive` (Akka) method. The use of conventional locks in the Scala or Java code are found by searching for the `synchronized` keyword. In this section we will investigate the occurrences of the different patterns found in the corpus and evaluate them using the properties as described in Sec. 8.1.2.

8.1.3.1. Locks

Similar to Java, all Scala object instances are associated with an intrinsic lock that can be used for synchronization with `synchronized` blocks. A synchronized block in Scala is always invoked on some object (when no receiver is specified it is invoked implicitly on the `this` pseudo-variable). All synchronized blocks synchronized on the same object can only have one actor executing inside them at the same time. All other actors attempting to enter the synchronized block are blocked until the actor inside the synchronized block exits that block. The lock used by the block is acquired at the start of the block and released at the end. While analyzing the code examples found in the corpus we have identified two ways programmers synchronize access to shared state using synchronized blocks. On the one hand, the synchronization can be done on the server-side. In that case typically (part of) the methods of the interface through which the shared state is accessed are synchronized. On the other hand, the synchronization can be done on the client-side. In that case it's the client's responsibility to acquire the lock before accessing the shared resource. To distinguish between objects that use server-side and client-side locking, the following metric was used: if the target of the synchronized block is local to the object invoking that block, then that object acts as the *server-side* interface through which the shared resource needs to be accessed. If the target of the synchronized block is not local to the object, then that object is a *client* that requires synchronous access to the shared resource. The advantages and disadvantages of both server-side and client-side locking will be discussed in this section.

Server-side locking
Throughout this section a simple toy example is used to illustrate the different patterns. In the example, the resource that is being shared is a simple integer value with a getter and a setter. In each example many clients will concurrently increase the value of the integer. To avoid race conditions, every `increase` operation needs to be serialized.
Fig. 8.1 illustrates how to model such a shared counter using a server-side lock. The only method that is publicly accessible is the `increase` method and that method is serialized by using the `synchronized` keyword.
While analysing the corpus 166 examples of the use of server-side locking were encountered. In most cases (116/166) access to the shared resource was synchronized by making some or all of the methods of the interface to the shared resource synchronized. In that case, the target of the synchronized block is the server-side object. If the internal shared resource (in our case the integer value) is not exposed to the clients then each access to the shared resource passes via the server-side interface and is thus serialized. In other examples (34/166) the data structure or container that represents the shared resource was used as the target of the synchronized block. That allows for a more fine-grained locking

```scala
class Server {
  private var c: Int = 0
  private def get(): Int = c
  private def set(n: Int) {
    c = n
  }
  def increase() = synchronized {
    set(get + 1)
  }
}
```

```scala
class Client(s: Server) extends Actor {
  def act {
    s.increase
  }
}
```

Figuur 8.1.: A server-side lock in Scala.

strategy where only part of each method invocation is synchronized. In some cases (16/166) an explicit lock was used by using the intrinsic lock of a newly created dummy object. This is usually the case when the server-side object synchronizes access to multiple shared resources and using the server-side object as the target is too fine-grained.
The advantage of managing the synchronization on the server-side is that synchronization is **enforced** when the shared resource is accessed on the client-side. A malicious or poorly written client cannot introduce race conditions when accessing the shared resource as each access to the shared resource is synchronized by the server. The disadvantage of using this approach is that if the server-side does not expose the lock that is used, then a client cannot **compose** different server-side operations into a larger synchronized operation.

Client-side locking

Fig. 8.2 illustrates how to model the shared counter using a client-side lock. If all the clients agree upfront to use a specific lock before accessing the shared resource then each access to the shared resource will be serialized.

```scala
class Server {
  private var c: Int = 0
  def get(): Int = c
  def set(n: Int) {
    c = n
  }
}
```

```scala
class Client(s: Server) extends Actor {
  def act {
    s.synchronized {
      s.set(s.get + 1)
    }
  }
}
```

Figuur 8.2.: A client-side lock in Scala.

While analysing the corpus, 38 examples of the use of client-side locking were encountered. In most cases (35/38) the intrinsic lock of the server object is used. In a few cases (3/38) an explicit lock is used by using the intrinsic lock of a newly created dummy object. This approach is mostly used when clients can hold a reference to a nested object of the shared resource. In that case an explicit lock needs to be used to ensure that all accesses to all nested objects of the shared resource are synchronized.
The advantage of managing the synchronization on the client-side is that the client can arbitrarily **compose** operations on the shared resource in a larger synchronized operation. The downside to this approach is that there is no way to **enforce** this type of synchronization. A single client that does not acquire the lock before accessing the shared resource can introduce a race condition for every other client.

Project	**Total**	**Server**	**Client**
CIMTool	4	4	0
SCADS	25	22	3
ThingML	9	7	2
bigbluebutton	20	15	5
blueeyes	1	1	0
diffa	14	8	6
ensime	4	1	3
evactor	1	1	0
gatling	4	4	0
geotrellis	6	6	0
kevoree	10	9	1
scalatron	1	1	0
signal-collect	3	2	1
socko	1	1	0
spark	99	82	17
spray	2	2	0
Total	204	166	38

Tabel 8.2.: Locks in the projects

Conclusion

Tab. 8.2 summarizes the results of our survey concerning locking mechanisms. In 80% of the cases synchronization is done on the server-side. Which suggests that **enforced** synchronization is usually more important than **composability** of different operations.

A general advantage of using locks over other non-blocking synchronization mechanisms is that the client-side does not need to **apply CPS** when accessing the shared resource. Each access to the shared resource blocks until it yields a result. However, the downside is the potential introduction of **deadlocks** when two different locks are nested. In the entire corpus, no occurrences of nested locking were found and while the results of our survey do not allow us to make any hard claims as to the reason why, avoiding nested locking is a good technique to avoid potential deadlocks. Through its interoperation with Java, Scala has access to Java's `java.util.concurrent.locks.ReentrantReadWriteLock` for allowing **parallel reads** of a shared resource. However, only a single occurrence of that type of lock was found in the entire corpus which might indicate that programmers do not value parallel reads as an important optimization. Using a lock in Scala does not **enforce isolation** as any access to a leaked reference to a nested object will not be synchronized.

8.1.3.2. Delegate Actor

Scala has two actor libraries, namely Scala Actors and Akka (See Chapter 2). In our case study we investigated projects that use either of those libraries. However, in both cases the use of a delegate actor as a synchronization pattern was found. Because of how a delegate actor is used, it's always a mechanism to use for client-side synchronization. Two predicates were used for distinguishing delegate actors from regular actors. Firstly, does the interface of the actor directly translates to the interface of the shared resource? Secondly, does the communication between the client actor and the delegate actor happen in a request-response style and is there is no communication with other actors involved?
Fig. 8.3 illustrates how the delegate actor pattern can be used in Scala to synchronize access to a shared resource. All `Increase` messages sent by different clients are serialized by the inbox of the server actor. The benefit of using a delegate actor over locks is that using asynchronous communication is a non-blocking synchronization mechanism and thus avoids **deadlocks**. Please note that Scala does have support for synchronous communication with an actor. However, when using exclusively synchronous communication with a delegate actor the benefit of using actors over locks is lost. In this section we only consider delegate actors in combination with asynchronous communication. One of the disadvantages of using delegate actors is that the client-side code needs to **apply a CPS transformation** when the result of a message is needed. Clients are not able to perform **read-only operations in parallel** because each opera-

```scala
class Server extends Actor {
  private var c: Int = 0
  private def get(): Int = c
  private def set(n: Int) {
    c = n
  }
  def act {
    loop {
      react {
        case Increase =>
          set(get + 1)
      }
    }
  }
}
```

```scala
class Client(s: Server) extends Actor {
  def act {
    s ! Increase
  }
}
```

Figuur 8.3.: A delegate actor in Scala.

tion on the shared resource is serialized by the inbox of the delegate actor. Synchronization is **enforced** because the client-side is forced to use the message-passing protocol to access the shared resource. However, in contrast with other actor languages, Scala does not provide actor isolation. Thus, any leaked reference to a nested object in the actor is not protected and can be freely accessed. Using the delegate actor pattern in Scala does not protect the **entire object graph** of the shared resource. Two messages sent by the same client can be interleaved with messages sent by other clients. There is no way for a client to put extra synchronization conditions on batches of messages. This means that the client cannot **compose** several smaller operations into a larger synchronized operation.
Tab. 8.3 summarizes the results of our survey concerning the use of actors in the corpus. In slightly over half of the cases (57/102) an actor is used to model a software entity that models some part of the application logic and is involved in communication with several other actors. In some cases (28/102) the actor was used as a delegate actor. In a minority of the cases (17/102) the actor served some other purpose such as doing some logging or forwarding messages. From these results we can conclude that in some cases the delegate actor pattern is used by software developers as a synchronization mechanism in favor of other mechanisms such as locks.

Project	**Total**	**Regular**	**Delegate**	**Other**
CIMTool	4	1	3	0
SCADS	3	0	0	3
ThingML	2	1	0	1
bigbluebutton	9	6	2	1
blueeyes	10	0	5	5
diffa	2	1	1	0
ensime	5	3	2	0
evactor	12	6	3	3
gatling	4	4	0	0
geotrellis	5	5	0	0
kevoree	0	0	0	0
scalatron	1	1	0	0
signal-collect	3	1	1	1
socko	9	3	5	1
spark	18	11	5	2
spray	15	14	1	0
Total	102	57	28	17

Tabel 8.3.: Actors in the projects

	No CPS	**Deadlock free**	**Parallel reads**	**Enforced Synchro-nization**	**Composable Interface**	**Enforced Isolation**
Server-side Lock	✓	×	✓	✓	×	×
Client-side Lock	✓	×	✓	×	✓	×
Delegate Actor	×	✓	×	✓	×	×

Tabel 8.4.: The different synchronization patterns and their properties

8.1.4. Conclusion

Tab. 8.4 summarises the different synchronization patterns found in the corpus and their properties. When choosing a synchronization mechanism there is always a consideration between a blocking and a non-blocking mechanism. On the one hand, a non-blocking synchronization mechanism such as the delegate actor requires the client-side to apply a CPS transformation when the result of an operation on the shared resource is required. On the other hand, blocking synchronization mechanisms such as locks can potentially introduce deadlocks. Another consideration to be made is whether or not the synchronization mechanism is applied on the client or the server-side. On the one hand, the advantage of server-side synchronization is that the synchronization is enforced. Any client that tries to access the shared resource is forced to synchronize its access. On the other hand, client-side synchronization allows the client to compose various operations on the shared resource in a larger synchronized operation. Executing read-only operations in parallel is an optimization that can be done using read-write locks. However, only a single occurrence of such a lock was found in the corpus. When using the delegate actor pattern it is impossible to execute read operations in parallel because they are serialized by the inbox of the delegate actor. Because a lock cannot be associated with an entire object graph, isolation of the shared resource cannot be guaranteed. In pure actor systems, isolation is guaranteed. However, Scala actors do not guarantee isolation.

8.2. A Shared Domain Library for Scala

To validate the claim that the synchronization patterns found in the previous sections can be replaced with domain abstractions, a library was written for Scala. Unfortunately, no amount of library code we can add to our implementation is going to guarantee that the different domains are fully isolated. When representing a shared resource, developers can still choose the path of least resistance and circumvent the use of domain references. This weakens the overall guarantees of the library. Another issue with the library approach is that it is impossible to distinguish between read-only and read-write methods. Without access to the underlying VM, it is impossible to *trap* assignments at runtime. Because an implementation of observable domains would require us to be able to trap assignments at runtime, at this time, only a library for shared domains was implemented. In SHACL every object is owned by the lexically enclosed domain. Because the Scala reflection API does not give access to that kind of information a more explicit approach for denoting object ownership was chosen. When using our library the convention is that any object that is shared among Scala actors needs to be "tagged" as a domain reference. For distinguishing between read-only and read-write methods without having to trap assignments, a similar technique was used. Our technique uses higher-order functions as proxies to intercept method invocations with minimal syntactic overhead.
Lst. 8.1 illustrates how to use the domain library for Scala. Every object instance that is eligible for sharing between different actors needs to define what domain it belongs to. In the example the `Server` class extends the `DomainReference` trait. That trait has one abstract field namely `domain` that needs to be defined for every instance of a `DomainReference`. On line 2 we assign a new domain to the `domain` field of our shared resource. If synchronization was needed over

```
1  class Server extends DomainReference {
2    val domain = new Domain
3    private var c: Int = 0
4    def get(): Int = reader { c }
5    def set(n: Int) = writer {
6      c = n
7    }
8  }
9
10 class Client(s: Server) extends DomainActor {
11   def act {
12     whenExclusive(s) {
13       s.set(s.get + 1)
14     }
15   }
16 }
```

Listing 8.1: A domain reference in Scala.

different instances of the `Server` class then the instantiation of the domain can be externalized. However, in our example the `Server` class is meant to be instantiated only once so it's fine to instantiate the domain together with the server object. Because the Scala reflection API does not allow us to *trap* assignments at runtime there is no way to distinguish between read-only and read-write methods. To solve this issue a number of runtime *tags* were added to the library to allow the programmer to make this distinction. In the example, every method of the `Server` class is tagged as being a `reader` or a `writer` method. Note that the code block after the tag is treated as an anonymous function, while `reader` and `writer` take that block as call-by-name parameter, which is only evaluated if the appropriate view is acquired. If an actor does not have the appropriate view at the time of executing the method, a runtime exception is thrown. Programmers should follow the convention to make any instance variable or untagged method private to avoid unsynchronized access. In our example the `get` method is tagged as a reader and the `set` method is tagged as a `writer`.
On the client-side, any actor that wants to access a domain reference should extend the `DomainActor` trait which itself extends the Scala Actor trait. Only a `DomainActor` can request a view on a domain. On line 12 an exclusive view is requested on the domain of `s`. Once the domain becomes available for exclusive access, a notification is scheduled in the inbox of the `Client` actor.

8.2.1. Shared Domains for Scala: The Implementation

The implementation of the `DomainReference` trait is given in Lst. 8.2. The DomainReference trait has an undefined field domain. Any concrete instance of a class that extends the trait will have to provide a value for `domain`. To intercept method invocations on domain references two methods were added as tags. The `reader` and `writer` methods accept a single no-parameter function (the body of the method) as a call-by-name parameter. Each of them checks if the current actor has a shared or exclusive view on the domain of the DomainReference. If the actor does not have the required view, a runtime exception is thrown.

```scala
trait DomainReference {
  val domain: Domain
  def reader[T](body: => T): T = {
    if(domain.hasReadAccess(Actor.self))
      body
    else throw DomainException("No read access to domain")
  }
  def writer[T](body: => T): T = {
    if(domain.hasWriteAccess(Actor.self))
      body
    else throw DomainException("No write access to domain")
  }
}
```

Listing 8.2: The `DomainReference` trait.

The implementation of the `DomainActor` trait is given in Lst. 8.3. The `receive` and `react` methods of the original actor trait are overridden such that messages that are sent to a `DomainActor` are intercepted. The implementation of `react` is similar to `receive` and was intentionally left out. If a notification is received, the notification's closure is executed before any other messages are processed. The first parameter of the `whenShared` and `whenExclusive` primitives is the domain reference on which a view needs to be acquired. The second parameter is a code block that needs to be executed once the domain of the domain reference becomes available for shared or exclusive access respectively. The return value of both primitives is a Scala future that represents the return value of executing the block. Executing `whenShared` or `whenExclusive` is an asynchronous operation. The view request is scheduled with the domain of the domain reference and the future is returned immediately. That future will eventually be resolved with the return value of executing the view. The implementation of the `whenExclusive` primitive is similar to the `whenShared` primitive and is intentionally left out.

```scala
trait DomainActor extends Actor {
  override def receive[R](body: PartialFunction[Any, R]): R = {
      val wrapper: PartialFunction[Any, R] = {
        case Notification(closure) =>
          closure()
          receive(body)
        case any =>
          body(any)
      }
      super.receive(wrapper)
    }

  override def react[R](body: PartialFunction[Any, R]): R = { ... }

  def whenShared[T](domainReference: DomainReference)(view: => T): Future[T] = {
    val p = promise[T]
    domainReference.domain.requestShared(Request(Actor.self, () => {
      p success view
    }))
    p.future
  }

  def whenExclusive[T](domainReference: DomainReference)(view: => T): Future[T] = {...}
}
```

Listing 8.3: The `DomainActor` trait.

8.2.2. Properties of the Domain Model

	No CPS	Deadlock free	Parallel reads	Enforced Synchronization	Composable Interface	Enforced Isolation
Server-side Lock	✓	×	✓	✓	×	×
Client-side Lock	✓	×	✓	×	✓	×
Delegate Actor	×	✓	×	✓	×	×
Shared Domains	×	✓	✓	✓	✓	✓

Tabel 8.5.: The different synchronization patterns and their properties

Tab. 8.5 summarizes the different synchronization mechanisms, including Shared Domains, and their properties. Because the `whenExclusive` is an asynchronous primitive, the client will need to **apply CPS** if the result of the `whenExclusive` block is needed. However, inside a view, the actor has unlimited synchronous access to the objects owned by the domain and no CPS is needed. Because every operation of the domain model is non-blocking, using domains as a synchronization mechanism remains **deadlock free**. Any number of **read-only** operations can be executed in **parallel** by requesting a shared view. The **synchronization is enforced** as any attempt to access a domain reference outside of a view will result in a runtime error. During a view the client-side actor can **compose** any number of operations in a larger synchronous operation. Because synchronization is enforced on a per-domain basis rather than per-object, **isolation** of the whole domain is enforced.

8.2.3. Pattern Transformation to Scala Library

This section shows how to transform any of the synchronization mechanisms and patterns shown in Sec. 8.1.3 by using the domains library. For each pattern an example was chosen among the code examples found during the survey. The examples and how they were transformed to the domain library can be found in Appendix A.

8.2.3.1. Delegate Actor

Lst. A.1 shows an instance of the delegate actor found in the *spark* project. From the comments associated with the code it is clear that the developers chose an actor as a synchronization mechanism to guarantee deadlock freedom. The translation of the delegate actor pattern into domains is a straightforward one and preserves deadlock freedom. In Lst. A.2, the `LocalActor` class now extends the `DomainReference` class instead of the `Actor` class. Please note that this also means that instances of this class will no longer spawn a new actor. On line 6, a new domain is created and is associated with the LocalActor. If we wanted to achieve full isolation of the shared resource and all of its nested objects, that domain could be associated with those nested objects. The behavior description of the actor (the `receive` statement) was translated into method definitions. Each method was tagged with the `writer` tag as none of the methods were read-only. The client-side code is not shown here. However, each asynchronous message can be trivially translated into first acquiring an exclusive view and during the view synchronously invoking the corresponding method.

8.2.3.2. Server-side lock

Lst. A.3 shows an instance of a shared resource in the *signal-collect* project where the synchronization is done on the server-side. The shared resource is a `Map` that maps names to actor systems. Each method of the interface of the ActorSystemRegistry is synchronized. The transformation to domains is shown in Lst. A.4. The `ActorSystemRegistry` now extends the `DomainReference` trait and each of the methods is tagged with either the `reader` or `writer` tag depending on whether the method is read-only or not. Please note that before the transformation, the `ActorSystemRegistry` did not make a distinction between read-only and read-write methods. By switching to domains we can identify read-only operations and allow parallel reads. On line 3, a new domain is created and is associated with the registry. On the client-side, which is not shown here, each of the clients needs to acquire a view before invoking any of the methods on the `ActorSystemRegistry`.

8.2.3.3. Client-side lock

Lst. A.5 shows an instance of client-side synchronization in the *diffa* project. The `handleMismatch` method is executed on the client and the `writer` is the shared resource that needs to be sychronized. In the example the `handleMismatch` method uses the intrinsic lock of the `writer` to ensure that the `clearUpstreamVersion` method is synchronized. Lst. A.6 shows the transformation of the example using domains. Instead of using the intrinsic lock of the `writer` an exclusive view on the domain of the writer is requested (line 9). That request returns a future. On line 17 we register a closure with that future, using `onSuccess`, to retrieve the return-value of the request, namely the `correlation`. Please note that the `onSuccess` method is an asynchronous operation and thus the `handleMismatch` method will now also be asynchronous where it previously was a synchronous method (Unless we use `Await` to ensure that the future becomes completed). This means that CPS will need to be applied to each method that calls handleMismatch and requires its result to be synchronously applied.

8.2.3.4. Conclusion

For delegate actors, the transformation to domains is straightforward because both synchronization mechanisms are non-blocking. In the case of locks, the transformation can only be applied if the client-side code does not require an immediate result.

In the translation of the examples to domains there is always only a single object associated with each domain. Currently, in Scala, full isolation of the whole object graph of a shared resource cannot be enforced with either locks or delegate actors. For that reason the translation of the examples does not have that property either. However, if we would associate each object of the shared resource with the same domain we would get that property as well.

8.3. Discussion

With the survey we have identified several code patterns for synchronizing access to a shared resource. The survey was conducted with a purely syntactical analysis of the code. While this approach was sufficient for identifying the different patterns, it does lack precision in some cases.
With the survey we have shown that developers use both client-side and server-side synchronization mechanisms. Unfortunately the analysis did not identify cases when both approaches are combined. For example, because Scala locks are reentrant, it is possible to use the same lock on the client-side as on the server-side to get some of the benefits of both approaches.
The survey does not identify which of the programs are distributed over the network. If a shared resource is distributed over the network then that could influence the choice of synchronization mechanism.
Using a delegate actor has the additional benefit that queries to the shared resource are computed in parallel with the client-side code. If the client does not require the result of the query immediately then it can do some other computations while the server-side computes the result of the query. With the syntactical analysis we are unable to ascertain whether or not the delegate actor was used purely as a synchronization mechanism or also as an optimization.
Scala has methods of concurrency other than actors such as futures. Shared domains are a synchronization mechanism tailored towards the actor model and in its current inception does not interact favorably with futures. A domain reference can only be accessed from within a `DomainActor`.

8.4. Conclusion

A survey of existing Scala projects was conducted to identify the different code patterns used by developers to synchronize access to a shared resource. That survey has shown that developers mix different synchronization mechanisms based on application-specific demands. We have shown that the domain model has a number of desirable properties for combining both server-side and client-side synchronization and have shown a possible Scala implementation for shared domains. This chapter also gives an overview on how to transform the different synchronization patterns found in the survey to our domain library.

9 CONCLUSION

This dissertation presents the domain model as an abstraction for object heaps in view of synchronizing access to a shared resource within the Actor Model in a safe, expressive way. This final chapter summarizes the advantages of our model, gives an overview of the contributions of this dissertation and highlights some directions for future work.

9.1. Summary

This dissertation presents the domain model as a synchronization mechanism for shared state in modern actor systems. The domain model allows safe and expressive state sharing among actors. The advantages of domains over other traditional synchronization mechanisms is that it is integrated with the actor model in such a way that it can provide the same safety guarantees as the original model.

Safety With the introduction of the domain model actors are no longer strictly isolated software entities. However, the domains themselves are 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 principle which is important for formal reasoning about program semantics, and provides additional guarantees to facilitate application development. The isolated turn principle guarantees that any turn can be regarded as a single isolated operation. That means that developers can make operations on domains as large or small as needed without potentially introducing deadlocks or race conditions.

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 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.

9.2. Contributions

This section gives an overview of the different chapters in this dissertation and their specific contributions.

- Chapter 2 gives an overview of a selection of state of the art actor systems and classifies them according to four distinct families of the actor model. The four families are: the original actor model, processes, active objects and communicating event-loops. In addition to the categorization of the different actor systems this chapter gives an overview of the different properties of specific actor systems and ties those back to their respective family. This chapter also defines the *isolated turn principle* which is an key concept throughout this dissertation.

- Chapter 3 gives an overview of the different synchronization mechanisms that can be used to synchronize access to a shared resource in both pure and impure actor systems. This chapter shows that impure actor systems are often flexible when it comes to representing shared state but compromise on overall safety. Programmers have to rely on synchronization mechanisms provided by the underlying shared-memory model such as locks or software transactional memory. However, these are not always well integrated with the actor model itself and lead to known concurrency issues such as deadlocks and race conditions. Pure actor systems provide more safety guarantees but are more stringent and thus compromise on expressiveness. A shared resource needs to be either encapsulated in a delegate actor or replicated over the different actors in the system. In this chapter we show that there is a need for a synchronization mechanism that is well integrated with the Actor Model.

- Chapter 4 gives an overview of the communicating event-loop model and illustrates why it has some desirable properties over other actor models when it comes to designing modular, reusable, secure, and fault-tolerant systems. In contrast with other actor models, the communicating event-loop model was designed for coarse-grained concurrency. The fact that each actor can export multiple interfaces as distinct behaviors allows for modular software design. This chapter also gives an overview of the SHACL programming language as a research tool for designing language abstractions. SHACL is an imperative, prototype-based programming language with communicating event-loop actors as its main abstraction for concurrency.

- Chapter 5 presents the core contribution of this dissertation, the domain model. We present the taxonomy that led to the design of four distinct types of domains, namely immutable, isolated, observable and shared domains. We give an overview of the different types of domains and their instantiation in SHACL. We show for each individual type of domain that it maintains the guarantees of the original pure actor model with regards to deadlock freedom, race condition freedom, and the isolated turn principle. This chapter illustrates that domains have some desirable properties over other synchronization mechanisms and relates our approach to some of the related work.

- Chapter 6 gives an operational semantics that serves as a precise specification of the semantics of the domain model. This chapter starts by giving an operational semantics for a core subset of the SHACL programming language without the domain model. We show that unifying object heaps, that come from the original model, with isolated domains maintains the semantics of the original model. This calculus is extended with new semantic rules for adding the three other types of domains and we show that the unification of object heaps and domains allows for a clean specification of this extension.

- Chapter 7 introduces domain handlers as a generic implementation strategy for enforcing the semantics of the different types of domains. A domain handler defines a number of traps that serve as callbacks for the interpreter. Each domain handler can then specify the behavior of a certain domain type by specializing the default traps.

- Chapter 8 validates the domain model by means of a survey of a relevant set of existing open source Scala projects. The survey lists the different synchronization mechanisms and patterns used by the developers of the projects for synchronizing access to a shared resource. We give an overview of the advantages and disadvantages of each pattern. We also show that developers mix synchronization patterns depending on what properties are desirable for each specific use-case. An implementation of shared domains in Scala is given and we show that the different patterns found during the survey can be transformed to domains. We show that the domain model has some desirable properties over the other synchronization mechanisms.

9.3. Future Work

This section outlines several possible avenues for future work. First, while we have shown that the domain model has a number of desirable properties when combined with the communicating event-loop model, it remains to be investigated if the domain model can be successfully combined with other concurrency models. Second, the current implementation of the domain model in SHACL put more emphasis on semantics than performance. Another interesting avenue for future work would be to investigate additional optimization techniques. Third, the implementation and specification of other domains can be investigated by making domain handlers first class. Last, the specification of the different properties of the domain model have remained largely informal and those properties could be strengthened by a formal proof using the operational semantics.

9.3.1. Domains for Other Concurrency Models

Domains for Other Families of Actor Languages

The domain model was originally designed for the communicating event-loop model. Sec. 8.2 shows that a translation to a language in the *process* family is possible but the translation has a number of limitations. For example, in contrast with the CEL model, in the process model a single turn is not a well defined concept. In our implementation we defined a turn as all the statements executed between two receive statements. Where a turn starts and ends strongly depends on the control flow of the program. That limits our understanding of the guarantees the domain model provides as it works on turn boundaries. We have not yet investigated how the translation of the domain model to actor systems that model other actor models such as the *original actor model* or the *active objects* model would work.

Synchronous Domains

Some actor systems (e. g., Scala Actors) allow different actors to communicate synchronously. This has the disadvantage that deadlock freedom can no longer be guaranteed. However, synchronous communication has the benefit that CPS no longer needs to be applied on the client-side. In the current domain model every view request is an asynchronous operation and the view is executed in a later turn of the actor that issued the request. Another avenue for future work would be to investigate how the properties of the domain model would be influenced when considering a synchronous version of the model.

Domains for Other Concurrency Mechanisms
The domain model is designed to synchronize access to a shared resource within the communicating event-loop model. However, the idea of isolating part of the heap in a separate software entity for synchronization has been applied for other concurrency mechanisms (See Sec. 5.9) as well. One of the most important properties of the domain model is that it guarantees the isolated turn principle. This means that one of the first challenges to make the domain model useful for other concurrency models would be to define the notion of a turn in those models.

9.3.2. Performance

Performance Evaluation
Each of the types of domains in our model solves the same set of issues while maintaining the guarantees of the original pure actor model. For each type of domain this involves a different synchronization mechanism. The developer has to choose a mechanism based on the use case and the acceptable performance trade-off. One of the main points for future work is determining that performance trade-off and comparing the overhead of domains to traditional approaches.

Static Type System
The current implementation of SHACL, as discussed in Chapter 7 is an abstract syntax tree interpreter for a dynamically typed language. The current implementation uses *domain handlers* to dynamically dispatch field accesses, method invocations and message sends to the appropriate domain handler. Another avenue for future work would be to investigate whether a static type system could help reduce the dynamic overhead of these calls. Also, the current version throws a runtime error when a domain is illegally accessed by an actor. A static type system could help with determining these errors at compile-time.

A Better Multi-Version History STM
The current implementation of observable domains uses a multi-version history STM to ensure that the different actors always see a consistent snapshot of the observable domain. The current implementation of the STM is a naive implementation that stores all the versions of each object. Another part of the future work is to investigate whether known techniques in that field [Perelman et al., 2010] can be used to optimize this implementation.

Integration of domains with the runtime To improve performance we could integrate the implementation of the domain model with the runtime by using the memory protection mechanism of the underlying operating system or VM for efficiently intercepting writes. A similar technique was used in Hoffman et al. [2011] to leverage the existing memory protection mechanism of the hardware, avoiding the need for inline security checks or write barriers.

9.3.3. Other

First-class Domain Handlers

The current implementation of domain handlers as discussed in Chapter 7 treats domain handlers as second-class objects. We have already shown that domain handlers can be used to implement domains with various other properties. Another potential avenue for future work would be to investigate how a first-class treatment of domain handlers can even further improve their flexibility and allow the specification of other domain types.

Formal Proofs for Properties of the Domain Model

The evaluation of the different properties of the domain model in this dissertation remain largely informal. The current version of the operational semantics does not serve any other goal than to provide a specification of the semantics of our model. These operational semantics could be used to provide proofs for each of the properties of the model.

9.4. Closing Conclusion

Historically, concurrent and parallel programming were mainly used as a specialist technique for systems programming or high performance computing. However, during recent years, commodity hardware from desktops to smartphones are being transformed into multi-core machines. This has led to an increased interest in more advanced concurrent programming models that are able to exploit the heterogeneous concurrency of desktop applications with more complex interactions between its different components. In a larger context, the focus of this dissertation is about the contrast between flexible concurrency control and manageable concurrency control that aims to guarantee safety and liveness. The starting hypothesis of this dissertation is that developers of complex interactive applications benefit most from a concurrency model with high safety and liveness guarantees. Throughout this dissertation we have shown that the actor model is a model that provides many desirable properties for designing modular, reusable, secure and fault-tolerant software. Several mainstream languages, such as Clojure [Hickey, 2008] and Scala [Haller and Odersky, 2007], are adopting the actor model as a concurrency model. 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 have shown that strict isolation between the different processes is not a necessary property to provide these guarantees. This dissertation presents the domain model as a set of language abstractions for controlling access to shared mutable state. 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. 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.

A

SCALA PATTERN TRANSFORMATION

A.1. Delegate Actor

URL: https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/scheduler/local/LocalBackend.scala

```scala
/**
 * Calls to LocalBackend are all serialized through LocalActor. Using an actor makes the calls on
 * LocalBackend asynchronous, which is necessary to prevent deadlock between LocalBackend
 * and the TaskSchedulerImpl.
 */
private[spark] class LocalActor(
  scheduler: TaskSchedulerImpl,
  executorBackend: LocalBackend,
  private val totalCores: Int) extends Actor with ActorLogReceive with Logging {

  val executor = new Executor(
    localExecutorId, localExecutorHostname, scheduler.conf.getAll, isLocal = true)

  override def receiveWithLogging = {
    case ReviveOffers =>
      reviveOffers()

    case StatusUpdate(taskId, state, serializedData) =>
      scheduler.statusUpdate(taskId, state, serializedData)
      if (TaskState.isFinished(state)) {
        freeCores += scheduler.CPUS_PER_TASK
        reviveOffers()
      }

    case KillTask(taskId, interruptThread) =>
      executor.killTask(taskId, interruptThread)

    case StopExecutor =>
      executor.stop()
  }

  def reviveOffers() {
    val offers = Seq(new WorkerOffer(localExecutorId, localExecutorHostname, freeCores))
    for (task <- scheduler.resourceOffers(offers).flatten) {
      freeCores -= scheduler.CPUS_PER_TASK
      executor.launchTask(executorBackend, task.taskId, task.name, task.serializedTask)
    }
  }
}
```

Listing A.1: A delegate actor

A.2. Delegate Actor Transformation

```
private[spark] class LocalActor(
  scheduler: TaskSchedulerImpl,
  executorBackend: LocalBackend,
  private val totalCores: Int) extends DomainReference {

  val domain = new Domain
  val executor = new Executor(
    localExecutorId, localExecutorHostname, scheduler.conf.getAll, isLocal = true)

  def reviveOffers() = writer {
    val offers = Seq(new WorkerOffer(localExecutorId, localExecutorHostname, freeCores))
    for (task <- scheduler.resourceOffers(offers).flatten) {
      freeCores -= scheduler.CPUS_PER_TASK
      executor.launchTask(executorBackend, task.taskId, task.name, task.serializedTask)
    }
  }

  def statusUpdate(taskId, state, serializedData) = writer {
    scheduler.statusUpdate(taskId, state, serializedData)
      if (TaskState.isFinished(state)) {
        freeCores += scheduler.CPUS_PER_TASK
        reviveOffers()
      }
  }
  def killTask(taskId, interruptThread) = writer {
    executor.killTask(taskId, interruptThread)
  }
  def stopExecutor() = writer {
    executor.stop()
  }
}
```

Listing A.2: Transformation of the delegate actor pattern

A.3. Server-side Lock

URL: https://github.com/uzh/signal-collect/blob/master/src/main/scala/com/signalcollect/configuration/ActorSystemRegistry.scala

```scala
object ActorSystemRegistry {

  var systems = Map[String, ActorSystem]()

  def register(system: ActorSystem) {
    synchronized {
      systems += ((system.name, system))
    }
  }

  def contains(system: ActorSystem): Boolean = {
    synchronized {
      systems.contains(system.name)
    }
  }

  def remove(system: ActorSystem) {
    synchronized {
      systems -= system.name
    }
  }

  def retrieve(name: String): Option[ActorSystem] = {
    synchronized {
      systems.get(name)
    }
  }
}
```

Listing A.3: A server-side lock

A.4. Server-side Lock Transformed

```
object ActorSystemRegistry extends DomainReference {

  val domain = new Domain

  var systems = Map[String, ActorSystem]()

  def register(system: ActorSystem) = writer {
    systems += ((system.name, system))
  }

  def contains(system: ActorSystem): Boolean = reader {
    systems.contains(system.name)
  }

  def remove(system: ActorSystem) = writer {
    systems -= system.name
  }

  def retrieve(name: String): Option[ActorSystem] = reader {
    systems.get(name)
  }
}
```

Listing A.4: Transformation of the server-side lock

A.5. Client-side Lock

URL: https://github.com/lshift/diffa/blob/6fe49c01c777fab37832b2c13763f9dbe58a7c44/kernel/src/main/scala/net/lshift/diffa/kernel/differencing/BaseScanningVersionPolicy.scala

```scala
def handleMismatch(scanId:Option[Long],
                   pair:PairRef,
                   writer: LimitedVersionCorrelationWriter,
                   vm:VersionMismatch,
                   listener:DifferencingListener) = {
  vm match {
    case VersionMismatch(id, attributes, lastUpdate, usVsn, _) =>

      val corrrelation = writer.synchronized {
        if (usVsn != null) {
          writer.storeUpstreamVersion(VersionID(pair, id), attributes, lastUpdate, usVsn, scanId)
        } else {
          writer.clearUpstreamVersion(VersionID(pair, id), scanId)
        }
      }

      handleUpdatedCorrelation(corrrelation)
  }
}
```

Listing A.5: A client-side lock

A.6. Client-side Lock Transformed

```
def handleMismatch(scanId:Option[Long],
                   pair:PairRef,
                   writer: LimitedVersionCorrelationWriter,
                   vm:VersionMismatch,
                   listener:DifferencingListener) = {
  vm match {
    case VersionMismatch(id, attributes, lastUpdate, usVsn, _) =>

      val f = whenExclusive(writer) {
        if (usVsn != null) {
          writer.storeUpstreamVersion(VersionID(pair, id), attributes, lastUpdate, usVsn, scanId)
        } else {
          writer.clearUpstreamVersion(VersionID(pair, id), scanId)
        }
      }

      f onSuccess {
        case correlation =>
          handleUpdatedCorrelation(corrrelation)
    }
  }
}
```

Listing A.6: Transformation of the client-side lock

REFERENCES

N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised5 report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, September 1998. ISSN 0362-1340. doi: 10.1145/290229.290234. URL `http://doi.acm.org/10.1145/290229.290234`. 48, 50

Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5. 11, 13

Gul Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9): 125–141, September 1990. ISSN 0001-0782. doi: 10.1145/83880.84528. URL `http://doi.acm.org/10.1145/83880.84528`. 11, 13, 14

Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, January 1997. ISSN 0956-7968. doi: 10.1017/S095679689700261X. URL `http://dx.doi.org/10.1017/S095679689700261X`. 30

Jamie Allen. *Effective Akka*. O'Reilly Media, Inc., 2013. ISBN 1449360076, 9781449360078. 4, 21, 41

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X. 3, 16, 33, 46, 84

E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359715. URL `http://doi.acm.org/10.1145/359636.359715`. 1

Mark Astley. The actor foundry: A java-based actor programming environment, 1998-99. URL `http://osl.cs.uiuc.edu/foundry`. 3

Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2006. URL http://www-sop.inria.fr/oasis/proactive/doc/ProgrammingComposingDeploying.pdf. 3, 19, 85

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. URL http://doi.acm.org/10.1145/209937.209958. 1

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. *SIGPLAN Not.*, 44(10):97–116, October 2009. ISSN 0362-1340. doi: 10.1145/1639949.1640097. URL http://doi.acm.org/10.1145/1639949.1640097. 83

Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. doi: 10.1145/343477.343502. URL http://doi.acm.org/10.1145/343477.343502. 34

Jean-Pierre Briot. Actalk: a testbed for classifying and designing actor languages in the smalltalk-80 environment. pages 109–129. University Press, 1989. 14

Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, April 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.004. URL http://dx.doi.org/10.1016/j.ic.2008.12.004. 18

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094852. URL http://doi.acm.org/10.1145/1103845.1094852. 83

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. 11

Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems Structures*, (0):–, 2014. ISSN 1477-8424. doi: http://dx.doi.org/10.1016/j.cl.2014.05.002. URL `http://www.sciencedirect.com/science/article/pii/S1477842414000335`. 87, 112

Joeri De Koster. The shacl programming language. `http://soft.vub.ac.be/~jdekoste/shacl/`, 2014. 116

Joeri De Koster and Tom Van Cutsem. Shacl: Operational semantics. Technical report, Vrije Universiteit Brussel, 2013. `http://soft.vub.ac.be/Publications/2013/vub-soft-tr-13-26.pdf`.

Joeri De Koster, Stefan Marr, and Theo D'Hondt. Synchronization views for event-loop actors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 317–318, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145873. URL `http://doi.acm.org/10.1145/2145816.2145873`.

Joeri De Koster, Tom Van Cutsem, and Theo D'Hondt. Domains: Safe sharing among actors. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions*, AGERE! '12, pages 11–22, New York, NY, USA, 2012b. ACM. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414644. URL `http://doi.acm.org/10.1145/2414639.2414644`.

Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. Tanks: Multiple reader, single writer actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 61–68, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541331. URL `http://doi.acm.org/10.1145/2541329.2541331`. 70

W. De Meuter, S. Gonzalez, and T. D'Hondt. The design and rationale behind pico. Technical report, Vrije Universiteit Brussel, 1999. URL `ftp://prog.vub.ac.be/pub/Courses/CPL/PDF.dir/PicoRationale.pdf`. 48

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492. 1

Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D&#39;Hondt, and Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35726-2, 978-3-540-35726-1. doi: 10.1007/11785477_16. URL http://dx.doi.org/10.1007/11785477_16. 21

Brian Demsky and Patrick Lam. Views: Object-inspired concurrency control. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 395–404, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806858. URL http://doi.acm.org/10.1145/1806799.1806858. 84

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, September 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90014-7. URL http://dx.doi.org/10.1016/0304-3975(92)90014-7. 91

Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6. 121

Google. The go programming language. http://golang.org/, 2011. Accessed: 2014-09-15. 116

Olivier Gruber and Fabienne Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 281–301, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8_12. URL http://dx.doi.org/10.1007/978-3-642-39038-8_12. 23, 86

Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72793-4. URL http://dl.acm.org/citation.cfm?id=1764606.1764620. 4, 21, 41, 156

Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.09.019. URL http://dx.doi.org/10.1016/j.tcs.2008.09.019. 42

Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1884002. 4, 86

C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977. ISSN 00043702. doi: 10.1016/0004-3702(77)90033-9. URL http://dx.doi.org/10.1016/0004-3702(77)90033-9.

C. Hewitt and B. Smith. A plasma primer (draft), 1975. 12

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624775.1624804. 2, 8, 11, 12

Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-270-2. doi: 10.1145/1408681.1408682. URL http://doi.acm.org/10.1145/1408681.1408682. 156

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, August 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL http://doi.acm.org/10.1145/359576.359585. 2

Kevin J. Hoffman, Harrison Metzger, and Patrick Eugster. Ribbons: A partially shared memory programming model. *SIGPLAN Not.*, 46(10):289–306, October 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048091. URL http://doi.acm.org/10.1145/2076021.2048091. 83, 155

Gregor Hohpe. Programmieren ohne stack: Ereignis-getriebene architekturen. *OBJEKTspektrum*, 2:18–24, 2006. ISSN 0945-0491. 36

S. Rougemaille 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`. 3

Myeong-Wuk Jang. The actor architecture manual, March 2004. URL `http://osl.cs.uiuc.edu/aa`. 3

Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. 36

Dennis Kafura. Act++: Building a concurrent c++ with actors. *J. Object Oriented Program.*, 3(1):25–37, April 1990. ISSN 0896-8438. URL `http://dl.acm.org/citation.cfm?id=90482.90493`. 14

Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709. 48

Wooyoung Kim. Thal: An actor system for efficient and scalable concurrent computing, 1997. 14

Joeri De Koster, Stefan Marr, Theo D'Hondt, and Tom Van Cutsem. Domains: Safe sharing among actors. *Science of Computer Programming*, 98, Part 2 (0):140 – 158, 2015. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2014.02.008. URL `http://www.sciencedirect.com/science/article/pii/S0167642314000495`. Special Issue on Programming Based on Actors, Agents and Decentralized Control.

Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. URL `http://dx.doi.org/10.1109/MC.2006.180`. 2, 41, 79

Mohsen Lesani and Antonio Lain. Semantics-preserving sharing actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 69–80, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541332. URL `http://doi.acm.org/10.1145/2541329.2541332`. 85

Microsoft. Axum programming language, 2008-09. URL `http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx`. 85

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing*, TGC'05, pages 195–229, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-30007-4, 978-3-540-30007-6. URL http://dl.acm.org/citation.cfm?id=1986262.1986274. 3, 19, 45

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. 1

Stas Negara, Rajesh K. Karmani, and Gul Abdulnabi Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 81–90. ACM, 2011. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941566. 86

Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835704. URL http://doi.acm.org/10.1145/1835698.1835704. 72, 123, 154

Constantine Plotnikov. Asyncobjects framework. http://asyncobjects.sourceforge.net/, 2007. 3

Mike Rettig. Jetlang, 2008-09. URL http://code.google.com/p/jetlang/. 3

Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1883996. 87

Christophe Scholliers, Eric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, February 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.03.011. URL http://dx.doi.org/10.1016/j.scico.2013.03.011. 38, 86

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224987. URL http://doi.acm.org/10.1145/224964.224987. 2, 71

Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_6. URL `http://dx.doi.org/10.1007/978-3-540-70592-5_6`. 3, 22

D.C. Sturman and G.A. Agha. A protocol description language for customizing failure semantics. In *Reliable Distributed Systems, 1994. Proceedings., 13th Symposium on*, pages 148–157, Oct 1994. doi: 10.1109/RELDIS.1994.336900. 14

Gerald Jay Sussman and Guy L Steele. Scheme: An interpreter for extended lambda calculus, 1975. 11

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`, 2005. 1

H. Sutter. Welcome to the jungle. `http://herbsutter.com/welcome-to-the-jungle/`, 2011. 5

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8_13. URL `http://dx.doi.org/10.1007/978-3-642-39038-8_13`. 132

C. Tomlinson, W. Kim, M. Scheevel, V. Singh, B. Will, and G. Agha. Rosette: An object-oriented concurrent systems architecture. In *Proceedings of the 1988 ACM SIGPLAN Workshop on Object-based Concurrent Programming*, OOPSLA/ECOOP '88, pages 91–93, New York, NY, USA, 1988. ACM. ISBN 0-89791-304-3. doi: 10.1145/67386.67410. URL `http://doi.acm.org/10.1145/67386.67410`. 14

David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0. doi: 10.1145/38765.38828. URL `http://doi.acm.org/10.1145/38765.38828`. 51

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, SCCC '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3017-6. doi: 10.1109/SCCC.2007.4. URL `http://dx.doi.org/10.1109/SCCC.2007.4`. 3, 21

Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, December 2001. ISSN 0362-1340. doi: 10.1145/583960.583964. URL `http://doi.acm.org/10.1145/583960.583964`. 3, 17, 33

Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming abcl/1. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 258–268, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28722. URL `http://doi.acm.org/10.1145/28697.28722`. 15, 39, 54

William Zwicky. Aj: A systems for buildings actors with java, 2008. 4