Applying Domains in Practice: A Case Study in Scala

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

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

Abstract

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 report 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 report is concluded by showing that the domain model has a number of desirable properties over the other synchronization mechanisms found in the survey.

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

1. Shared State Synchronization Patterns: A Scala Survey

A recent study [**?**] 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.

1.1. The Corpus of Actor Programs

For our case study we reuse the corpus of [**?**] 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.

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

[1] http://actor-applications.cs.illinois.edu/index.html

[2] https://github.com

- **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. Table 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.

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

Table 1: The corpus of projects used in the survey

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 report is to evaluate the different synchronization patterns found in the survey according to these properties. Throughout this report 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.

[3] http://cloc.sourceforge.net/

- **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.

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 Section 1.2.

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.

Figure 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.

```
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)
  }
}
```

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

Figure 1: A server-side lock in Scala.

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 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.
Figure 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.

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

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

Figure 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

Table 2: Locks in the projects

Conclusion.
Table 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.

1.3.2. Delegate Actor

Scala has two actor libraries, namely Scala Actors and Akka (See **??**). 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?

Figure 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.

```
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)
      }
    }
  }
}
```

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

Figure 3: A delegate actor in Scala.

Clients are not able to perform **read-only operations in parallel** because each operation 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.

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

Table 3: Actors in the projects

Table 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.

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

Table 4: The different synchronization patterns and their properties

1.4. Conclusion

Table 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.

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.

listing 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 different instances of the `Server` class then the

```
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 1: A domain reference in Scala.

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.

2.1. Shared Domains for Scala: The Implementation

The implementation of the `DomainReference` trait is given in listing 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.

The implementation of the `DomainActor` trait is given in listing 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

```
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 2: The `DomainReference` trait.

the view. The implementation of the `whenExclusive` primitive is similar to the `whenShared` primitive and is intentionally left out.

```
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 3: The `DomainActor` trait.

2.2. Properties of the Domain Model

Table 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.

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

Table 5: The different synchronization patterns and their properties

2.3. Pattern Transformation to Scala Library

This section shows how to transform any of the synchronization mechanisms and patterns shown in Section 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 **??**.

2.3.1. Delegate Actor

?? 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 **??**, 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.

2.3.2. Server-side lock

?? 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 **??**. 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`.

2.3.3. Client-side lock

?? 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. **??** 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.

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.

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`.

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 report also gives an overview on how to transform the different synchronization patterns found in the survey to our domain library.