Crème de la Crem: Composable Representable Executable Machines

Architectural Pearl

Marco Perone
Treviso, Italy
pasafama@gmail.com

Georgios Karachalias
Tweag
Paris, France
georgios.karachalias@tweag.io

Abstract

In this paper we describe how to build software architectures as a composition of state machines, using ideas and principles from the field of Domain-Driven Design. By definition, our approach is *modular*, allowing one to compose independent subcomponents to create bigger systems, and *representable*, allowing the implementation of a system to be kept in sync with its graphical representation.

In addition to the design itself we introduce the *Crem* library, which provides a concrete state machine implementation that is both compositional and representable. *Crem* uses Haskell's advanced type-level features to allow users to specify allowed and forbidden state transitions, and to encode complex state machine—and therefore domain-specific—properties. Moreover, since *Crem*'s state machines are representable, *Crem* can automatically generate graphical representations of systems from their domain implementations.

CCS Concepts: • **Hardware → Finite state machines**; • **Software and its engineering → Software design engineering**.

Keywords: domain architecture, domain-driven design, state machine

1 Introduction

Tactical Domain-Driven Design focuses on identifying transactional boundaries and is often combined with an event-based architecture [Stopford 2018], which is based on the flow of messages (e.g. commands and events) throughout an application domain and the clear separation of responsibilities among several components (e.g. aggregates, policies and projections).

Such separation often allows to fruitfully discuss with non-technical domain experts the details of the inner workings of the domain itself and to translate them directly into working code. Moreover, as the understanding of the domain deepens as time passes, the architecture itself allows for refactorings towards deeper insights.

Still, terms and concepts like aggregates, policies and projections lack a precise definition and delimitation, and this tends to create discussions and confusion.

Another issue we experienced developing systems using Domain-Driven Design techniques is the distance which appears between the theoretical model developed through distilling knowledge from the domain experts and the concrete implementation of such a model. This is basically the same issue which happens with stale documentation, where the documentation of a piece of code is not up-to-date with the current behaviour of the code itself.

With the work described in this paper we try to mitigate these issues bringing together ideas from Domain-Driven Design, state machines and functional programming, expressing DDD and event-based architectures in terms of state machines using Haskell.

The main novel contributions of this paper are:

- Using state machines to implement policies and projections, building on top of the existing knowledge of treating aggregates as state machines [Ploch 2022].
  This allows to use state machines as the unique underlying concept needed to implement a whole application domain.
  Moreover, such an approach makes it all compositional, since state machines compose extremely well.
- Implementing state machines, and therefore architectures based on them, in such a way that information could be extracted from the implementation and used as documentation of the system.
  Specifically, we are able to generate a graphical representation out of the domain implementation which describes the relevant information about the domain itself. This is extremely helpful when discussing the behaviour of the domain with domain experts, removing the need for understanding and explaining the bare code behaviour.
  The ability of generating a graphical representation out of a state machine was already available with *motor* [Wickström 2019]. While *motor* is focused on building single state machines, this article and the *Crem* library improve the situation introducing compositionality.
- Making explicit the connection between an abstract model and its concrete implementation via a novel

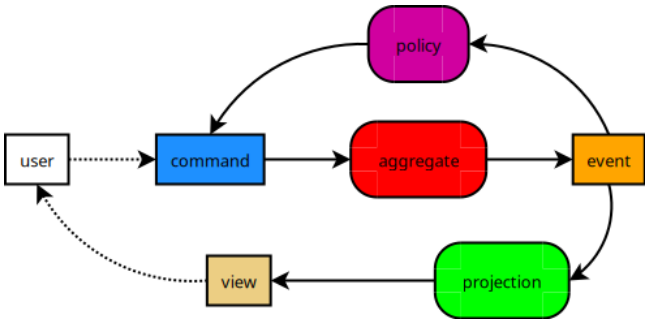

Figure 1. Domain-Driven Design Model. Arrow direction denotes the flow on information. Solid arrows describe software interactions, while dotted ones refer to human ones. Square boxes represent data types, while rounded ones represent processes.

combination of Domain-Driven Design ideas and state machines.

- Using Haskell's powerful type system to encode complex properties of the implemented state machines, and therefore of the domains they represent.
  With *Crem* we are able to express explicitly which state transitions are allowed and which, on the other hand, are forbidden.
  Using Haskell we are able to get access both to features mimicking dependent types, needed for the whole machinery to work, and also to the whole ecosystem of production-ready libraries and frameworks.

For example, with *Crem* we can create systems, with an architecture based on Domain-Driven Design principles, just by composing state machines as follows

```
wholeCartDomain :: StateMachine CartCommand [CartView]
wholeCartDomain = Kleisli
  (Feedback cart paymentGateway)
  paymentStatus
```

and generate a graphical representation describing the implemented system (see Figure 7).

2 Domain-Driven Design

Domain-driven design, as a community and as a practice, aims to build software for complex domains creating a shared understanding of the domain and expressing it in a model, which describes the problem space [Verraes 2021].

One popular shape of such models, which emerges naturally from Event Storming workshops [Brandolini 2015], is rooted in ideas such as Aggregates [Evans 2003], Event Sourcing [Young 2007][1] and CQRS [Young 2010] and is captured in Figure 1.

[1]Notice that the architecture that we describe in the paper is very well suited for Event Sourcing, but does not rely on it and is actually independent of how persistence is dealt with

The flow of such an architecture goes as follows:

- A user expresses their desired interaction with the software with a *Command*, comprising all the necessary information to execute that command.
- The *Command* is received by an *Aggregate*, which has the role of making sure that every invariant of the system is preserved. Then, it emits its decisions as *Events*.
- *Events* describe the relevant state transitions of the system, containing all the data necessary to propagate information through the system and potentially to reconstruct the current state of the system.
- *Policies* describe the reactive logic of the application. Whenever an *Event* is emitted by an *Aggregate*, the system might decide to react to it emitting a new *Command*.
- *Projections* aggregate the information contained in the *Events* and condensate them into specialised *Views*.
- *Views* describe the information which are then shown to and used by the user to decide which *Command* they should require next.

We will henceforth use the colors used for defining the terms whenever there is an instance of this term. For example, as *Aggregate* is colored red[2], the *Cart* aggregate will also be red.

As a recurring example, we will consider the checkout process of a standard ecommerce systems. In such a context, a user emits a *PayCart* command instructing the system to process the payment of their cart. The *Cart* aggregate will check all the system invariants to ensure that the cart is in a state where it could actually be paid (e.g. it is not empty); if every invariant is satisfied, the aggregate will emit a *CartPaymentInitiated* event. A *PaymentGateway* policy will contact an external system to actually process the payment; if the payment is processed correctly, the policy will emit a *MarkCartAsPaid* command. The command will then be processed by the *Cart* aggregate, which will emit a *CartPaymentCompleted* event. The *PaymentStatus* projection will react to the *CartPaymentInitiated* and *CartPaymentCompleted* events to provide the user with a *CartState* view describing the current state of the payment.

Aggregates and projections are by their nature pure stateful processes, meaning that the only effect which can take place is state management. In particular their logic does not depend on the interaction with the external world. All the relevant information for making sensible decisions is contained either in their state or provided by the incoming messages.

On the other hand, policies are by nature impure, since they often deal with interacting with external systems.

The cycle which get created between aggregates and policies helps to split the write side of the domain logic between

[2]Traditionally in *EventStorming* aggregates are yellow, but it doesn't render well on paper, so we switched to red.

its pure and its effectful parts, increasing the testability of the system.

3 State machines

In the architecture described above, while commands, events and views are just simple data, on the other hand aggregates, policies and projections are stateful processes. As such, they could be implemented as state machines. By *state machine* we always mean a *Mealy machine* [Mealy 1955], which is composed of a stateful function and the current value for the state. Practically, it consists of the current value of the state and an action to emit an output determined from an input and the current value of the state, while being able to update the state.

In Haskell terms, one potential implementation could be the following.

```
data Mealy s a b = Mealy
  { initialState :: s
  , action       :: s → a → (b, s)
  }
```

State machines come with many practical benefits:

- They are extremely compositional and allow combining simple state machines into more complex ones in several ways.
  For examples, two machines could be composed sequentially, feeding the output of the first machines as inputs of the second.
  Or they could be executed in parallel, providing the inputs to both machines and collecting their outputs.
  Or they could be executed in alternative, providing either the input to the first machine or the input the second, executing the corresponding machine, and obtaining as output either the output of the first machine or the output of the second.
- They allow a graphical representation as state diagrams, which could help to understand how a machine actually works also for non-technical people.
- They can be implemented as a function depending just on the *input* and the *state*. The simplicity of this mental model helps the implementor considering all the possible cases which need to be considered and uncovering potential edge cases.

For example, the *Cart* aggregate of the previous section could be implemented along these lines; see Figure 2.

This is a very simplified version, which does not perform any sophisticated logic on duplicate or unexpected messages, but still it gives the idea of how the domain logic could be implemented.

In real systems, one will have to deal with multiple aggregates, handling different kinds of commands. Since there will be only one aggregate to deal with a given command, we can compose several aggregates in parallel to obtain a bigger component able to deal with multiple kinds of commands. Then, when we receive a command, we will need to route it to the correct component to be dealt with.

```
data CartCommand
  = PayCart
  | MarkCartAsPaid

data CartEvent
  = CartPaymentInitiated
  | CartPaymentCompleted

data CartState
  = WaitingForPayment
  | InitiatingPayment
  | PaymentComplete

cart :: Mealy CartState CartCommand [CartEvent]
cart = Mealy
  { initialState = WaitingForPayment
  , action = \case
      WaitingForPayment PayCart →
        ([CartPaymentInitiated], InitiatingPayment)
      WaitingForPayment MarkCartAsPaid →
        ([], WaitingForPayment)
      InitiatingPayment PayCart →
        ([], InitiatingPayment)
      InitiatingPayment MarkCartAsPaid →
        ([CartPaymentCompleted], PaymentComplete)
      PaymentComplete PayCart →
        ([], PaymentComplete)
      PaymentComplete MarkCartAsPaid →
        ([], PaymentComplete)
  }
```

Figure 2. Cart Implementation, Simplified

Similarly, we can compose multiple policies or projects to put ourselves back in the case when we have a single policy or projection.

4 Crem

The architecture we described in the Domain-Driven Design section could be implemented as a composition of state machines. Aggregates, policies and projections get implemented as state machines, which then get composed into a unique state machine which describes the whole domain of the application.

To benefit from the above described perks of both the architecture and state machines we need a way to implement a domain with state machines which is compositional and allows synchronising a graphical representation of the state machines with the actual implemented code.

Crem is a Haskell library for defining and executing state machines in such a way that the theoretical benefits of state

machines, as compositionality and representability, are practically preserved.

It allows to:

- Compose state machines in multiple ways.
- Generate a graphical representation out of the implementation of a state machine.
- Impose invariants on the allowed transitions.

Crem is based on two main ideas: tracking the set of the allowed transitions at the type level and using a free-like structure to restore compositionality.

Let's see how these two ideas play really well together and why both have a relevant role in the implementation.

A simple definition of a state machine that is parametric over the type of its state s, as well as input and output types a and b, respectively, could look like the following:

```
data Mealy s a b = Mealy
  { initialState :: s
  , action       :: s → a → (b, s)
  }
```

By making the state type implicit, one ends up with *Mealy* as is defined in the *machines* library [Kmett et al. 2012]:[3]

```
newtype Mealy a b = Mealy
  { runMealy :: a → (b, Mealy a b)
  }
```

This allows improving the compositionality of the data type, since now there is no need to keep track of the state type variable, and it is possible to implement instances for common type classes like *Category*, *Profunctor* and *Arrow*.

On the other hand, these simple implementations do not allow to extract any information about how the state machine works, without actually executing the state machine itself. Being just functions, the only thing we can do with them is running them.

In particular, we have no way to extract information about which transitions are allowed or not by the state machine, and therefore we are not able to generate a graphical representation which describes how the machine works.

To be able to enforce which state transitions are actually allowed and to generate a graphical representation of the state space, we need to track the list of allowed transitions, which we call *Topology*.

```
newtype Topology vertex = Topology
  { edges :: [(vertex, [vertex])]
  }
```

The *Topology* is a list of edges, grouped by their initial vertex.

Coming back to our example, the *Topology* of the *Cart* aggregate could look as follows:

```
data CartVertex
  = WaitingForPaymentVertex
  | InitiatingPaymentVertex
  | PaymentCompleteVertex

cartTopology :: Topology CartVertex
cartTopology =
  [ (WaitingForPaymentVertex, [InitiatingPaymentVertex])
  , (InitiatingPaymentVertex, [PaymentCompleteVertex]
  , [PaymentCompleteVertex, []])
  ]
```

By storing the *Topology* at the type level we are able to use it for enforcing at compile time that forbidden transitions are never executed. Having an explicit *Topology* allows to retrieve the information later on to create a graphical representation of the state machine out of it. In this way, we are sure that the generated graphical representation is always in sync with the implemented logic of the state machine.

One possible way to implement such a mechanism is depicted in Figure 3.

AllowedTransition topology initialVertex finalVertex, which is implemented as in Figure 4, is a type class which checks that a transition from *initialVertex* to *finalVertex* is allowed by the *topology*. It searches through all the edges of the *Topology* whether there is one starting from *initialVertex* and ending at *finalVertex*, and it constructs a proof of it using the *AllowTransition* data type.

The *InitialState* and *ActionResult* data types are respectively ways to store a *state* and a pair of a *state* and an *output*, given the fact that the state does not have kind *Type* but kind *vertex* → *Type*.

With these new data types, the *Cart* aggregate could be adjusted accordingly. The updated implementation is shown in Figure 5.

Even though the implementation looks completely analogous to the previous version using the *Mealy* data type, we are now ensuring at compile time that the machine will never use a transition which is not allowed by the *Topology*.

The *BaseMachine* data type allows us to track at the type level the information regarding the *Topology* of the machine.

On the other hand, that *topology* type parameter makes composition harder, since composing two machines would require computing at the type level the topology of the composed machine. Moreover, just the presence of that third

[3] This transformation can be perceived as two separate steps:

1. First turn the s into an existentially quantified type variable:

```
data Mealy' a b = forall s. Mealy'
  { initialState :: s
  , action       :: s → a → (b, s)
  }
```

2. Then use recursion to eliminate the state variable altogether (see *unfoldMealy* from the *machines* library [Kmett et al. 2012]):

```
unfoldMealy :: Mealy' a b → Mealy a b
unfoldMealy (Mealy' initial action) = go initial where
  go s = Mealy $ \a → case action s a of
    (b, t) → (b, go t)
```

```
data InitialState (state :: vertex → Type) where
  InitialState :: state vertex → InitialState state

data ActionResult
     (topology :: Topology vertex)
     (state :: vertex → Type)
     (initialVertex :: vertex)
     output
  where
  ActionResult
    :: AllowedTransition topology initialVertex finalVertex
    ⇒ (output, state finalVertex)
    → ActionResult topology state initialVertex output

data BaseMachine
     (topology :: Topology vertex)
     input
     output = forall state.
  BaseMachine
  { initialState :: InitialState state
  , action
    :: forall initialVertex.
    state initialVertex
    → input
    → ActionResult topology state initialVertex output
  }
```

Figure 3. State Machines, Revisited

type parameter does not allow us to us common type classes as *Category*, *Profunctor* or *Arrow*.

To restore compositionality, we define a free-like structure which adds composition operations on top of the *BaseMachine* data type, which is presented in Figure 6.

This way, it becomes trivial to implement type classes like *Category*, *Strong* and *Choice*, in terms of the constructors.

The *Sequential* constructor allows restoring sequential categorical composition. The *Parallel* allows implementing *Arrow* and *Strong*, while the *Alternative* constructor allows implementing *ArrowChoice* and *Choice*.

The *Feedback* constructor is used to loop two state machines, respectively feeding the output of one as input of the other.

The *Kleisli* constructor allows composing sequentially two state machines which produce multiple outputs, while they process single inputs.

The *StateMachine* data type helps us construct an abstract syntax tree where the leaves are *BaseMachine*s and the other nodes describe how we are composing the subtrees.

Considering our running example, if we suppose now to have implemented the *Cart* aggregate, the *PaymentGateway*

```
data AllowTransition
     (topology :: Topology vertex)
     (initial :: vertex)
     (final :: vertex)
  where
  AllowIdentityEdge
    :: AllowTransition topology a a
  AllowFirstEdge
    :: AllowTransition ('Topology ('(a, b ': l1) ': l2)) a b
  AllowAddingEdge
    :: AllowTransition ('Topology ('(a, l1) ': l2)) a b
    → AllowTransition ('Topology ('(a, x ': l1) ': l2)) a b
  AllowAddingVertex
    :: AllowTransition ('Topology topology) a b
    → AllowTransition ('Topology (x ': topology)) a b

class AllowedTransition
     (topology :: Topology vertex)
     (initial :: vertex)
     (final :: vertex)
  where
  allowsTransition :: AllowTransition topology initial final

instance{-# INCOHERENT #-}
    AllowedTransition topology a a
  where
  allowsTransition = AllowIdentityEdge

instance{-# INCOHERENT #-}
    AllowedTransition ('Topology ('(a, b ': l1) ': l2)) a b
  where
  allowsTransition = AllowFirstEdge

instance{-# INCOHERENT #-}
    AllowedTransition ('Topology ('(a, l1) ': l2)) a b) ⇒
    AllowedTransition ('Topology ('(a, x ': l1) ': l2)) a b
  where
  allowsTransition = AllowAddingEdge allowsTransition

instance{-# INCOHERENT #-}
    AllowedTransition ('Topology topology) a b ⇒
    AllowedTransition ('Topology (x ': topology)) a b
  where
  allowsTransition = AllowAddingVertex allowsTransition
```

Figure 4. *AllowedTransition* implementation

policy and the *PaymentStatus* projection, like so

```
cart           :: StateMachine CartCommand [CartEvent]
paymentGateway :: StateMachine CartEvent [CartCommand]
paymentStatus  :: StateMachine CartEvent [CartView]
```

```
data CartState (cartVertex :: CartVertex) where
    WaitingForPayment :: CartState WaitingForPaymentVertex
    InitiatingPayment :: CartState InitiatingPaymentVertex
    PaymentComplete   :: CartState PaymentCompleteVertex

cart :: BaseMachine CartTopology CartCommand [CartEvent]
cart = BaseMachine
  { initialState = InitialState WaitingForPayment
  , action = \case
      WaitingForPayment PayCart →
        ActionResult ([CartPaymentInitiated], InitiatingPayment)
      WaitingForPayment MarkCartAsPaid →
        ActionResult ([], WaitingForPayment)
      InitiatingPayment PayCart →
        ActionResult ([], InitiatingPayment)
      InitiatingPayment MarkCartAsPaid →
        ActionResult ([CartPaymentCompleted], PaymentComplete)
      PaymentComplete PayCart →
        ActionResult ([], PaymentComplete)
      PaymentComplete MarkCartAsPaid →
        ActionResult ([], PaymentComplete)
  }
```

Figure 5. Cart Implementation, Revisited

we can compose them together to create an implementation for the whole domain:

```
wholeCartDomain :: StateMachine CartCommand [CartView]
wholeCartDomain = Kleisli
  (Feedback cart paymentGateway)
  paymentStatus
```

A whole implementation could be found in the *examples* folder inside the *Crem* repository [Perone 2023].

From this definition, *Crem* is able to generate a diagram that shows how the application is structured, presented in Figure 7.

From the picture, it's easy to understand how the *cart* aggregate is connected in a loop with the *paymentGateway* policy and how the outputs of the loop are then fed as inputs into the *cartState* projection.

Inside every box we can also see the topology of the state space of the state machine controlling that specific component.

When we interpret the *StateMachine* abstract syntax tree, we need to define how to process the leaves, i.e. the *BaseMachine*s, and how to interpret the composition of substrees.

For example, when we want to run a *StateMachine*, we need to describe how to run a *BaseMachine*, and this is provided by the *action* included in the *BaseMachine* definition, and how to run a composition of sub-*StateMachine*s. This depends on the specific constructor; for example, for the *Sequential* constructor, we first recursively run the first

```
data StateMachine input output where
  Basic
    :: forall m vertex topology input output.
       ( Demote vertex ~ vertex
       , SingKind vertex
       , SingI topology
       )
    ⇒ BaseMachine topology input output
    → StateMachine input output
  Sequential
    :: StateMachine a b
    → StateMachine b c
    → StateMachine a c
  Parallel
    :: StateMachine a b
    → StateMachine c d
    → StateMachine (a, c) (b, d)
  Alternative
    :: StateMachine a b
    → StateMachine c d
    → StateMachine (Either a c) (Either b d)
  Feedback
    :: StateMachine a [b]
    → StateMachine b [a]
    → StateMachine a [b]
  Kleisli
    :: StateMachine a [b]
    → StateMachine b [c]
    → StateMachine a [c]
```

Figure 6. State Machine Representation

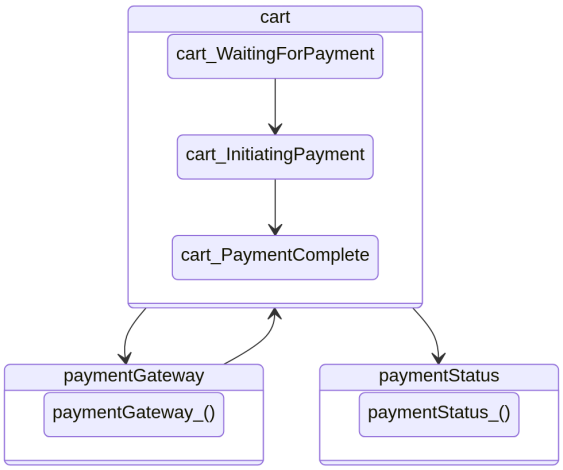

Figure 7. Architecture diagram of the cart payment system

StateMachine, and we use the output as input to run the second *StateMachine*.

Similarly, when we want to create a graphical representation of a *StateMachine*, we need to be able first to create a graphical representation of its leaves, which are *BaseMachine*s. We are able to do that because we have the *Topology* information stored at the type level. Then, we also need to be able to create a graphical representation of a composed *StateMachine* given a graphical representation of its sub-*StateMachine*s. Depending on the used constructor, and on the graphical representation that we would like to obtain, we can decide how to represent the composed machine.

5 Extending the architecture

In real-life projects an application is rarely composed by only one aggregate, one policy, and one projection. To understand how we can compose multiple aggregates, policies and projections together to model more complex workflows, we can use the interface provided by *Arrow* and *ArrowChoice*, or, alternatively, the *Profunctor*, *Strong*, and *Choice* type classes. Effectively, this means that the composition does not depend on our use of state machines; composition using another data type satisfying these type classes would also work in pretty much the same way.

Let's continue with our running example and consider that we would like to automatically start the delivery process once a payment is completed. As a first step, we introduce a new aggregate that we implement as a "Basic" state machine:

```
data ShippingCommand = ...

data ShippingEvent = ...

shipping :: StateMachine ShippingCommand [ShippingEvent]
shipping = Basic ...
```

To connect this new aggregate with the rest of the application, we want to execute it as an alternative to the current write model, meaning that we want to handle either a *CartCommand* or a *ShippingCommand* and route it to the appropriate machine. We can achieve this using the (+++) operator from *ArrowChoice* (or, equivalently, *splitChoice* from the *Choice* type class).

```
writeModelWithShipping :: StateMachine
  (Either CartCommand ShippingCommand)
  [Either CartEvent ShippingEvent]
writeModelWithShipping = rmap
  (fmap Left ||| fmap Right)
  (writeModel +++ shipping)
```

Now we have a way to process both commands concerning the cart payment and the shipping, but there is still no automatic connection between the two components. We would like to connect them by specifying that whenever a payment is completed the shipping process must start. The word *"whenever"* indicates that we need to use a policy, specifically one that consumes *CartEvent*s and produces *ShippingCommand*s.

```
paymentCompletePolicy
  :: StateMachine CartEvent [ShippingCommand]
paymentCompletePolicy = stateless $ \case
  CartPaymentInitiated   → []
  CartPaymentCompleted → [StartShipping]
```

We now need to connect our new *paymentCompletePolicy* policy to our *writeModelWithShipping* machine. Policies are always connected via the *Feedback* constructor of the *StateMachine* data type. To use it, we need to align the types by ignoring some inputs and enlarging the return type:

```
writeModelWithShipping' :: StateMachine
  (Either CartCommand ShippingCommand)
  [Either CartEvent ShippingEvent]
writeModelWithShipping' =
  Feedback writeModelWithShipping $
    rmap (fmap Right) paymentCompletePolicy
      ||| stateless (const [])
```

Next, we need another projection to provide the user with some queryable information about the status of the shipping. It is a process which consumes *ShippingEvent*s and produces some new *ShippingInfo* value:

```
shippingInfo :: StateMachine ShippingEvent [ShippingInfo]
```

The only thing that remains to be done is connecting the machine (*writeModelWithShipping'*) and the two projections (*paymentStatus* and *shippingInfo*). We first pair up the projections to set up our read model and then we link it to our write model in a sequential fashion:

```
readModel :: StateMachine
  (Either CartEvent ShippingEvent)
  [Either CartView ShippingInfo]
readModel = rmap
  (fmap Left  |||  fmap Right)
  (paymentStatus +++ shippingInfo)

cartAndShipping :: StateMachine
  (Either CartCommand ShippingCommand)
  [Either CartView ShippingInfo]
cartAndShipping =
  Kleisli writeModelWithShipping' readModel
```

6 Future work

Some possible directions for future work on the subject could be:

- The *Feedback* constructor of the *StateMachine* data type is used to implement looping behaviour. Classically, such behaviour is implemented in terms of the *ArrowLoop* or *Costrong* type classes. It would be interesting to investigate the relation between *Feedback* and such type classes, potentially removing the *Feedback*

constructor from the *StateMachine* data type and replacing it with a constructor more similar to the *loop* or *unfirst* functions coming from the aforementioned type classes.
- The *Sequential* and *Kleisli* constructors both deal with sequential composition of state machines. It would be interesting to find a way to unify them.
- With the current implementation, we are tracking at the type level only a list of the allowed state transitions. There is no connection between the allowed transitions and the inputs which trigger them. In other words, we are not restricting in any way the inputs that we could receive in a given state. It would be interesting to implement a way to express that in a certain state only certain inputs are valid.
  This would allow stating explicitly which are the expected inputs in a given state when implementing a state machine, avoiding the need to implement error handling for such unwanted cases.
- The current implementation of *Crem* does not take advantage of any concurrency or parallelism. In cases where a state machine is actually composed of multiple sub-machines in parallel, it could be interesting to allow running them on separate threads to improve the execution time.
- In the spirit of parallelising as much as possible a given machine, or anyway optimizing it in other ways, it could make sense to introduce an optimization step which would restructure the abstract syntax tree which constitutes a *StateMachine*. Simple optimizations could be given by the algebraic structure provided by the categorial structure; for example any identity machine composed sequentially could be removed; or it would be possible to use the distributive law between sequential and parallel composition to improve the parallelisability of a machine.

Another potential area of future development concerns how to test systems built with *Crem* and/or with the architecture described in the *Domain-Driven Design* section.

Some potential aspects to investigate in that area could be:

- Using a compositional library like *Crem*, the Domain-Driven Design architecture described above could be implemented as a single state machine receiving commands as inputs and emitting views as outputs.
  One way to test such a system would be to proceed with unit testing, treating the whole domain as a unit, feeding commands to it and asserting the expected state on the view observed by the user.
- Another way to test such a system would be to use property-based testing, generating commands randomly and making assertions on the invariants of the views.
  A potential direction on investigation could be using something like linear temporal logic to provide a language to express system invariants, along the lines to what has been done with Quickstrom [O'Connor and Wickström 2022].
- Pushing it even further, we can notice that our domain and the user (in fact, any external system interacting with our system) together form a cycle. Therefore, if we implement our user as a state machine (potentially an effectful or probabilistic one) receiving views and producing commands, we could close the loop and let our application run as long as we like. At this point we could use this new complete system to test the invariants of the domain.
- Another way in which such a system could potentially be tested would be to use languages like TLA+ or Alloy. A model could be exported using the information stored in the topology of the machine. Then it could be tested and verified using the chosen specification language.

7 Conclusion

We believe that compositionality and representability constitutes two extremely important aspects for the success of a software architecture. The former because it allows tackling simpler problems and then composing back their solutions to obtain solutions for more complex problems. The latter because it allows an easier interaction between business and domain experts and software developers.

Crem makes it easy to keep the graphical representation synchronised with the implementation, providing a reliable source of graphical documentation.

Inspired by Domain-Driven Design and thanks to the Haskell type system we were able to create the *Crem* library, which allows architecting systems in a composable and representable way, without sacrificing the developer experience while implementing such systems.

Moreover, the architecture we propose helps to clarify the role of the various components commonly used in a Domain-Driven Design architecture, describing more precisely which is their role inside the domain.

Acknowledgments

We would like to thank the reviewers of FUNARCH '23 for their constructive comments, and especially Michael Sperber for shepherding the editing process; our paper is all the better for their thorough feedback and suggestions.

The initial development of the *Crem* library happened while both authors were employed by Tweag.

The first author would also like to thank his former colleagues Richard Eisenberg, Alexis King, Sjoerd Visscher, Alexander Esgen, Nicolas Frisby and Daniele Palombi for the fruitful discussions and the feedback on the *Crem* library.

Moreover, he would like to thank his former colleague Alexei Drake for the fruitful collaboration in setting up a development environment for *Crem* using *Nix*.

References

Alberto Brandolini. 2015. *Introducing EventStorming.* Leanpub, 1321 Blanshard Street, Suite 301, Victoria, British Columbia, Canada, V8W 0B6.

Eric Evans. 2003. *Domain-Driven Design.* Addison-Wesley, Reading, MA.

Edward A. Kmett, Rúnar Bjarnason, and Josh Cough. 2012. machines: Networked stream transducers. https://hackage.haskell.org/package/machines

George H. Mealy. 1955. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal* 34, 5 (1955), 1045–1079.

Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. *PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2022), 1025–1038.

Marco Perone. 2023. A Cart example with Crem. https://github.com/marcosh/crem/tree/main/examples/Crem/Example/Cart

Thomas Ploch. 2022. DDD Aggregates: Processes, State Machines and Transducers. https://t-pl.io/ddd-aggregates-processes-state-machines-and-transducers

Ben Stopford. 2018. *Designing Event-Driven Systems.* O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Mathias Verraes. 2021. What is Domain-Driven Design. https://verraes.net/2021/09/what-is-domain-driven-design-ddd/

Oskar Wickström. 2019. motor: Type-safe effectful state machines in Haskell. https://hackage.haskell.org/package/motor

Greg Young. 2007. The Architecture of a Large Transaction System. https://www.infoq.com/interviews/Architecture-Eric-Evans-Interviews-Greg-Young/

Greg Young. 2010. CQRS Documents. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf