Anoma Research Topics | TECHNICAL REPORT

Intent Machines

Anthony Hart[a] **and D. Reusche**[a]

[a]Heliax AG

* *E-Mail: {anthony,d}@heliax.dev*

Abstract

Intents, abstract entities encapsulating a user's desires, are a promising approach to designing modern, complex decentralized financial systems. This paper presents a theoretical framework for abstracting intent-processing systems through the introduction of "intent machines", entities capable of processing user intents and transforming system state accordingly. Special cases of such machines implement mechanisms like automated auctions, barter systems, and counterparty discovery. We present intent machines abstractly as a coalgebra; a nondeterministic state transition function capturing the machine's dynamics. This allows us to consider the most general notions of equivalence, composition, and interaction. We then provide several examples of intent machines, including a toy example where states are natural numbers and intents are weighted relations, and more practical instantiations for creating barter systems, utilizing auction bids as intents to distribute resources within a network. This research lays the foundation for future investigations into the game-theoretic aspects of intent processing, where the abstract formulation is intended to serve as a minimal formulation required for such work.

Keywords: Intents; Intent Machines; Decentralized Finance; Coalgebra; Formalization; Smart Contracts; Automated System Design; Barter Systems

(Received: January 22, 2024; Published: February 9, 2024; Version: February 13, 2024)

Contents

1. Introduction

As decentralized financial systems grow in size and complexity there is a need to serve its users more robustly. Whenever such a system is used, there is a goal in the user's mind about what is to be accomplished. Some modern approaches to system design have abstracted out a user's intent into a notion called, appropriately, an "intent". Exactly what an intent is is often left vague, but, broadly, intents ought to encapsulate the desires and conditions that users, or agents operating on their behalf, wish to express within a system, particularly when trading or transferring assets. These intents can range from a trader's desire to swap assets at a favorable exchange rate, to a user's condition to release funds only upon the successful completion of a programmed task, to complex combinations of logical, temporal, and financial stipulations often found in smart contracts and automated marketplaces. In the context of a contract enforcing the existence of a sophisticated institution, such as voting systems or decentralized corporate-like entities, more ambitious intents may express desires related to fairness, efficiency, trade-offs, etc. These intents may be thought of, and sometimes are formulated, as logical formulas about the state of the network.

At the core of the concept of intent is the expression of user preferences guiding automated systems in the execution of complex tasks without direct human supervision. The interpretation and processing of such intents is a difficult task, both from a computational and conceptual standpoint due to the complex web of interactions and contradictions between different user desires. Outcomes ought to be determined by algorithms designed to honor users' wishes as faithfully as possible within the limits of the system's rules. The processing and fulfillment of intents require specialized machinery within blockchain infrastructure, not only to register and interpret the preferences but also to match, execute, or otherwise reconcile them with the current and future states of the system. Algorithms accomplishing this will be implemented by "solvers"; entities operating within a network that users can send intents to for execution. Such solvers, in different forms, would be at the center of many decentralized finance applications, such as automated bartering systems, auction mechanisms, counterparty discovery, etc.

This report attempts to formulate the infrastructure of intents in such a way as to abstract from engineering specifics, enabling us to study intent-processing systems in their most general form. We aim to develop a theoretical framework that illustrates the structure of "intent machines" that process intents and alter the state of a system. This concept is very abstract and encompasses things that are outside the context of finance or distributed systems. Different intent machines will have different notions of state and intent, but their structure should match a common pattern. Our objective is to distill the essence of intent-handling mechanics to explore how they can compose and interact.

This report begins with the discussion of intent machines in the abstract, where we lay out the fundamental components of such systems: their inputs, their state, and how they transform these elements over time. We formulate these using the abstract notion of a coalgebra. We then give a toy instantiation of an intent machine where intents are relations between the current state and the next state of the network, represented as a natural number. The solver, at each step, picks a new number for the next state in an attempt to maximize the number of satisfied relations in the batch of intents given at that step. The subsequent sections build on these foundations, exploring practical instantiations of intent machines for barter, where the state is the distribution of resources within a network and the intents are auction bids. Different versions of the bid machine are given based on different requirements for the solvers.

Ultimately, these foundations lay the stage for future Anoma work on the game theory of intent solvers. The hope is that this coalgebraic presentation offers a useful and minimal framework to start these investigations.

2. Intent Machines in the Abstract

We introduce the notion of an "intent machine" by starting with a maximally abstract definition and progressing to more concrete special cases. This section does not assume what an intent is. This allows later instantiation of intents as a variety of different things. Later in this report, intents will be instantiated as logical formulas that the state must satisfy, and later intents will be instantiated as bids in an auction. There is not necessarily a shared structure behind these instantiations, so this section will treat intents as merely an element of some set that our machine processes at each step. The goal of this section is not to formalize intents, but, rather, the processing of intents. The most abstract version, which we call a batch machine, takes a state, which is just the element of some set, S, and waits for an input batch B (intuitively, a batch of *intents*) for further processing. The guiding example for a state is that it should contain the total information describing a network in a single instant, though the goal of this section is to treat the subject more abstractly and we will not need to assume any structure on the state for now. Once it receives the batch, it then nondeterministically outputs a new state, along with an output batch produced from the old batch. The prototype instance assumes the output batch is a subset of the input batch that was actually processed, though this is not the case in general. Exactly what batches will be and how they should be processed is not enforced by the most general definition, but it will be enforced by special cases and used as a guiding intuition for the general case.

2.1. Intent Machines as Coalgebras. We begin with a maximally abstracted view of intent machines as a kind of coalgebra (Jacobs, 2017). We characterize them by their transition function, which will be a coalgebra over the

monad $\mathcal{M}$ in [1], parameterized by some notion N of nondeterminism, and a set B of intent batches. If S represents the machine state, we define $\mathcal{M}$ as

$$\mathcal{M}_B^N(S) := (N(S \times B))^B. \tag{1}$$

Any function of type $S \to \mathcal{M}_B^N(S)$ for a fixed S is a transition function and coalgebra over $\mathcal{M}_B^N$. $\mathcal{M}$ will denote $\mathcal{M}_B^N$, wherever N and B are clear from context. The join operation for the monad $\mathcal{M}$ is denoted by $\mu_{\mathcal{M}}$. That $\mathcal{M}$ is a monad is shown in Section 7.1. Other than the standard properties of a monad, we only assume that there is an additional parametrically polymorphic function (Pierce, 2002) called "sample" of type $\forall\alpha.N(\alpha) \to \alpha$. This function implements a sampling procedure whose details we will not specify.

A machine itself is a pair consisting of a current state and a transition function. This is not a full intent machine, so we will call it a "batch machine", and we will see that full intent machines are a special case of batch machines. Batch machines are themselves variants of Mealy machines (Bonsangue et al., 2008) where we've added nondeterminism and forced the inputs and outputs to be the same set, B.

One may wonder why we do not bundle the batch and the state, as we may rewrite the transition function type as the isomorphic $S \times B \to N(S \times B)$, now a coalgebra over N. This would be an incorrect interpretation, however, as the output batch is not fed back into the machine at each time step, only the state is. That is, given a batch machine, (s, m), and a stream of batches, bs : Stream B, we may run the machine, producing another stream of batches as an output defined by the following corecursive equation:

$$\begin{aligned} \text{eval } m\ s\ bs : \text{Stream } B := &\text{let } x : S \times B = \text{sample } (m\ s\ (\text{head } bs)) \\ &\text{in } \quad \pi_2\ x\ ::\ \text{eval } m\ (\pi_1\ x)\ (\text{tail } bs) \end{aligned} \tag{2}$$

Where $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are the projection functions out of a product.

With this level of abstraction, we may start characterizing the most generic notions of composition. The most obvious stems from the observations that coalgebras over a monad are arrows in a Klesli category, giving rise to a sequential notion of composition, the "Klesli composition" of two machine transition functions. Given two transition functions, m_1 and m_2 of type $S \to \mathcal{M}(S)$, we may define this notion of composition as follows;

$$m_1 \circ_{\text{Kleisli}} m_2 := \mu_{\mathcal{M}} \circ \mathcal{M}(m_1) \circ m_2 \tag{3}$$

digrammatically

$$S \xrightarrow{m_2} \mathcal{M}(S) \xrightarrow{\mathcal{M}(m_1)} \mathcal{M}^2(S) \xrightarrow{\mu_{\mathcal{M},S}} \mathcal{M}(S)$$

Where μ is the multiplication operation of the monad. In practice, what this will do is evaluate the input state on m_2 send the output batch to m_1. This essentially defines a basic notion of sequential composition. The output will be the output of m_1.

This notion of composition will be unsatisfactory in general due to the output batch of m_2 being thrown away. In general, we may provide a function to combine output batches, $u : B \times B \to B$, and generalize Kleisli composition as:

$$m_1 \circ_u m_2 := \lambda s\ b.\ \mu_N(N(\lambda s_1\ b_1.N(\lambda s_2\ b_2.(s_2,\ u\ b_1\ b_2))(m_1\ s_1\ b_1))(m_2\ s\ b)) \tag{4}$$

In practice, u will typically be the union of two sets.

We may generalize this further. If we introduce a function that processes batches based on an output batch, $f : B \times B \to B$, then we can update the input batch to m_1 instead of giving it a raw copy.

$$m_1 \circ_u^f m_2 := \lambda s\ b.\ \mu_N(N(\lambda s_1\ b_1.N(\lambda s_2\ b_2.(s_2,\ u\ b_1\ b_2))(m_1\ s_1\ (f\ b\ b_1)))(m_2\ s\ b)) \tag{5}$$

A typical use case might have f be something like a filtering function, such as the set difference operation. This would allow the second machine to receive only those intents that were not processed by the first machine.

These compositions still enforce a notion of sequential compositionality. Application-wise, we may think of both machines collaborating on the same batch of intents. m_2 does what it can to satisfy the batch, it then updates the state and m_1 does its best to satisfy the same batch given the state change m_2 made. An example where this might be useful is if we have a machine that ignores batches and performs some bureaucratic function, such as flipping a bit within the state. If we have another machine whose behavior varies based on this bit but does not change it, then Klesli composing the bureaucrat machine with the bit-sensitive machine will create a machine that flips between the two modes of the bit-sensitive machine at each step. Another use case may occur if the batches can be split into two sub-batches, $B \cong B_1 \times B_2$. If we have two machines, one that only works on the B_1 part and another that only works on the B_2 part, then we can use Klesli composition to create a machine that works on the whole B.

We may also combine the underlying states of two machines for a notion of parallel composition. If we have $m_1 : S_1 \to \mathcal{M}(S_1)$ and $m_2 : S_2 \to \mathcal{M}(S_2)$, then we can define $m_1 \times_u m_2 : S_1 \times S_2 \to \mathcal{M}(S_1 \times S_2)$ as

$$m_1 \times_u m_2 := \lambda(s_1, s_2)\ b.\ \mu_N(N(\lambda(s_1', b_1).\ N(\lambda(s_2', b_2).\ (s_1',\ s_2',\ u\ b_1\ b_2))\ (m_2\ s_2\ b))\ (m_1\ s_1\ b)) \tag{6}$$

While we may not want to enforce a sequential nature to the composition, we may provide separate filtering functions to herd different parts of the batches to different machines. Assuming we have $f_1 : B \to B_1$ and $f_2 : B \to B_2$, and further generalize the previous types to $m_1 : S_1 \to \mathcal{M}^N_{B_1}(S_1)$, $m_2 : S_2 \to \mathcal{M}^N_{B_2}(S_2)$, and $u : B_1 \times B_2 \to B$, we may define $m_1 \times_u^{f_1,f_2} m_2$ as follows.

$$\lambda(s_1, s_2)\ b.\ \mu_N(N(\lambda(s_1', b_1).\ N(\lambda(s_2', b_2).\ (s_1',\ s_2',\ u\ b_1\ b_2))\ (m_2\ s_2\ (f_2\ b)))\ (m_1\ s_1\ (f_1\ b))) \quad [7]$$

We may provide a similar composition in the case of coproducts. We may define $m_1 + m_2 : S_1 + S_2 \to \mathcal{M}(S_1 + S_2)$ as

$$m_1 + m_2 := [\mathcal{M}(\iota_1) \circ m_1,\ \mathcal{M}(\iota_2) \circ m_2] \quad [8]$$

where

$$f : X \to Z, g : Y \to Z \vdash [f,\ g] : X + Y \to Z \quad [9]$$

is the universal property of the coproduct, and $\iota_1 : X \to X + Y$ and $\iota_2 : Y \to X + Y$ are the injections/constructors of the coproduct. This form of composition will switch between different machines depending on the form of the context. If $\mathbb{B}$ denotes the type of booleans, then, note that $\mathbb{B} \times S \cong S + S$. Therefore, switching between different machines based on a single bit is a special case of this kind of composition. Since only one machine can run in a given transition, there's no ambiguity or flexibility in terms of handling batch inputs/outputs. However, if there is some way to filter batches B into two different types of batches with $f_1 : B \to B_1$ and $f_2 : B \to B_2$, and we can cast batches back into B with $c_1 : B_1 \to B$ and $c_2 : B_2 \to B$, then we can use these to split the batches between the different machines with.

$$m_1 +_{c_1,c_2}^{f_1,f_2} m_2 := [\lambda s.N(- \times -)(\iota_1, c_1) \circ m_1(s) \circ f_1, \lambda s.N(- \times -)(\iota_2, c_2) \circ m_2(s) \circ f_2] \quad [10]$$

Where $N(- \times -)$ is a somewhat abusive notation for the bi-functor map turning a pair of functions $A \to C$ and $B \to D$ into a function $N(A \times B) \to N(C \times D)$.

We may ask what it means for two machines to be equivalent. This can be done by relating transition functions to final coalgebras. Given the greatest fixed point of $\mathcal{M}$, $G_{\mathcal{M}} := \nu X.(N(X \times B))^B$, the final coalgebra is the (isomorphism) coalgebra $\text{fix} : G_{\mathcal{M}} \to (N(G_{\mathcal{M}} \times B))^B$. What makes it final is the existence of, for any other coalgebra c over the state S, a unique function $\text{beh}_c : S \to G_{\mathcal{M}}$ identified by the coalgebra homomorphism property;

$$\begin{array}{ccc} S & \xrightarrow{\text{beh}_c} & G_{\mathcal{M}} \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle \text{fix}} \\ \mathcal{M}(S) & \xrightarrow{\mathcal{M}(\text{beh}_c)} & \mathcal{M}(G_{\mathcal{M}}) \end{array} \quad [11]$$

$G_{\mathcal{M}}$ represents the full, branching future history of a machine. It includes all points of interaction, all inputs, all outputs, and all layers of nondeterminism as a single, essentially infinite data structure. It includes exactly and only the observable aspects of a machine's execution. We may use it to define a natural notion of observational equivalence through the notion of behavioral equivalence. That is, we will treat two machines, S_1, c_1 and S_2, c_2, as "the same machine" if they produce the same behavior, if $\text{beh}_{c_1}(S_1) = \text{beh}_{c_2}(S_2)$.

We may further derive from any particular coalgebra a kind of modal logic. This works by defining a logic of predicates over states (Kupke and Pattinson, 2011). We may have a predicate P and define, for example, $\Box P$ to mean that P will always hold for the future, $\diamond P$ to mean that P will eventually hold, etc. Our particular coalgebra complicates this by being both a labeled transition system (due to the inputs at each step) and being probabilistic, though there are existing modal logic constructions for these cases. Such an approach is most useful for characterizing the behavior we want to guarantee about the machine. For example, consider a machine that takes payments per intent, and the size of these payments dictates the effort the machine puts into satisfying intents. Such a statement should be characterizable via an appropriate modal statement. We may be able to give precise probabilities about the likelihood of satisfying an intent depending on the methods used, the intent in question, and payment. The details will heavily depend on the specifics.

2.2. Intents Discussion. From here, we can make the definition more specific to clarify the specifics of an intent machine. We set batches to be sets of intents, $B = \mathcal{P}(I)$, the powerset of some type of intent, I. This does not tell us much without knowing what intents are.

We formulate intents as a pair consisting of a transition function and a partial weighted predicate over state transitions. The guiding intuition for this formulation is to separate control from desire. In the prototypical example of an intent, it will express a desire for some resource in exchange for another. The first component expresses a partial state transition where the intent may create/destroy what it has control over. This aids in composing intents. The second component expresses a kind of weighted predicate over transitions. If the transition satisfies the intent, then it returns an element between 0 and 1, representing a kind of utility.

$$I := (S \to S) \times (S \times S \to \star \cup [0, 1]) \quad [12]$$

where $\star$ is a singleton set with one element, which we will also call $\star$. In the case that $\star$ is returned, we consider the intent to be unsatisfied. This specifies the core structure of intents in so far as they are relevant to solving; that is to finding/optimizing transitions that actually satisfy intents. Note that this S should be the same S as the coalgebra is using in practice, although this is not a theoretical requirement. One may have trouble making useful transition functions if this S differs from that in the coalgebra, and we will assume they are the same for the rest of this section.

The intuition is that the first function is a state transition the intent maker has control over. In a typical example of an intent where someone wants an A for a B, the intent maker has control over the B they want to trade. The transition function would simply delete the B the intent maker has from the state. The state is then unbalanced, giving leeway for changing resources to rebalance. The predicate expresses whether the intent is satisfied by any proposed transitions and, if it is, how satisfied the intent maker is.

We will typically have a solving component, solver : $S \times \mathcal{P}(I) \to N(S \times \mathcal{P}(I))$, attempting to find state transition functions that satisfy the intents. This will have the property that, for all $s : S$, $is : \mathcal{P}(I)$, and all s', is' in the support of solver$(s,\ is)$

1. $is' \subseteq is$,
2. $\forall i \in is'.\ \pi_2\ i\ s\ s' \neq \star$.

This guarantees that any returned transition will satisfy the set of intents the solver claims to be solving.

One question that emerges is the nature of intent composition. The goal of intent composition is to, in some way, simplify a collection of intents into a single intent. The main example is a situation where we have two intents, one expressing a want for a B in exchange for a C, and the other expressing a want for an A in exchange for a B. Between these, we have an A and a B, and we need a C to complete the exchange. If we commit to making this trade, then we can fuse these into a single intent of the form "wants an A for a C". With this understanding in mind, we can motivate compositions.

The transition functions of the intents can be composed directly as functions. The net function will involve both agents executing whatever they have control over.

Composing the predicate requires more thought. We must make a new predicate representing the desires of both intents. We may characterize all relevant notions of composition through a choice of a function $f : (\star \cup [0,1])^n \to \star \cup [0,1]$. We may desire to split this function into a function that handles the $\star$ cases, and a function $f : [0,1]^2 \to [0,1]$. This is not possible in general, but for many approaches this makes sense. It requires the binary function to cleanly generalize to n-ary versions. This works so long as the operation is associative. There are, ultimately, two canonical ways to handle the $\star$s. We may always return a $\star$ so long as a single one is present, or we may return a $\star$ only when there is no other option. The former is the obvious choice, as we want the composition to be dissatisfied if either of the composites are. As for the associative, normalization preserving operations, three non-trivial operations that may be considered commonplace are max, min, and multiplication.

Since composition is supposed to characterize a situation where the first party is already satisfied, whatever we choose should be equivalent to the predicate of the second party in the case that the first party is fully satisfied. This reduces the options to either multiplication or min. Further thought will be required to understand what things should be.

To summarize, we can define intent composition as

$$(t_1, p_1) \uparrow (t_2, p_2) := (t_1 \circ t_2, \lambda(s_i, s_o).\text{lift2M} * (p_1\ (t_2\ s_i, s_o))\ (p_2\ (t_1\ s_i, s_o))) \quad [13]$$

We've issued transition functions to the states of each predicate's input state. Each already lives in their own post-execution state. This forces both to live in the same post-execution state.

This idealization may not work in practice. It may be better to have a representation, R, with which we can replace the second component of an intent with $S \to R$, which takes the current state and returns a proposition representation. We'd also have an evaluation function, eval : $R \to S \to \star \cup [0,1]$, which takes a representation and a proposed state (filling in the variables of the proposition) and outputs a measure of satisfaction. Alternatively, we could assume that I is an abstract type, and simply assume the existence of a function compile : $I \to S \to R$. This is likely closer to what is needed in reality, as optimizing over black-box functions is not practically possible. We cannot, for example, take a derivative, or some discrete analog, of a completely black-box function. This would limit us to unstructured optimization algorithms like evolutionary methods or stochastic search. In reality, we should always be able to reason about the stated preferences of the intent, which requires a first-order representation, rather than a black-box function. Such a representation would likely come in the form of an encoding for, for example, a weighted CSP.

2.3. Decomposition. We may ask when decomposition is possible. Generally, to facilitate decomposition in a semantically meaningful way, we need to know a thing or two about intents. Let's consider the case where the state space is decomposable into $S \cong S_1 \times S_2$. We would like to take a transition function of type $m : S \to N(S \times B)^B$ and turn it into two transition functions of type $m_1 : S_1 \to N(S_1 \times B_1)^{B_1}$ and $m_2 : S_2 \to N(S_2 \times B_2)^{B_2}$, respectively. This can be accomplished with two functions, $f_1 : B_1 \to B$ and $f_2 : B_2 \to B$, that cast batches for each machine.

Additionally, we need functions $r_1 : B \to B_1$ and $r_2 : B \to B_2$ to get filter batches only from each machine; without these functions, the return batches of both machines would be the composite output batch. Using these, we'd have

$$m_1 := \lambda s.N(- \times -)(\pi_1, r_1) \circ m(s) \circ f_1 \quad [14]$$

$$m_2 := \lambda s.N(- \times -)(\pi_2, r_2) \circ m(s) \circ f_2 \quad [15]$$

The key question at this point is what f_1 and f_2 should be. Conceptually, they should split batches of intents into sets that care about the different parts of the state. That is, for every batch in the input history, it should be provable that;

$$B \cong \{i \mid \forall s_1, s_1' \in S_1, s_{2a}, s_{2a}', s_{2b}, s_{2b}' \in S_2 . \pi_2\ i((s_1, s_{2a}), (s_1', s_{2a}')) = \pi_2\ i((s_1, s_{2b}), (s_1', s_{2b}'))\} \quad [16]$$

$$+\{i \mid \forall s_{1a}, s_{1a}', s_{1b}, s_{1b}' \in S_1, s_2, s_2' \in S_2 . \pi_2\ i((s_{1a}, s_2), (s_{1a}', s_2')) = \pi_2\ i((s_{1b}, s_2), (s_{1b}', s_2'))\} \quad [17]$$

that is, each batch should be decomposable into a batch of intents that do not vary based on S_1, and a batch of intents that do not vary based on S_2. f_1 would then return the set of first things while f_2 would return the set of second things. This is necessary so that we may interpret each intent as either an intent over just S_1 or S_2.

It, of course, may be easier to establish the existence of an m_1, m_2, such that $m = m_1 \times m_2$, rather than decomposing m directly. This is likely the best way to approach decomposition using $\circ$ and $+$ as well.

2.4. Reputation Discussion. For our purposes, a reputation system is a method to weigh different intents. This allows us to give more or less importance to some intents. If an intent has a higher weight, then this should influence the solver to spend more effort to satisfy the intent.

The simplest way to implement a reputation system is to simply assert the existence of a function $r : I \to [0, \infty)$. This can be used to scale the desire of each intent.

We may seek to verify certain properties of the reputation system. For example, to ensure one cannot gain priority by spamming intents, we may have a function $\mathrm{id} : I \to A$, assigning an identity (the type of which I've just called A) to each intent. We may have a theorem stating something like

$$\forall a \in A . \forall s, s' \in S . \left(\sum_{i \mid \mathrm{id}(i) = a \wedge \pi_2\ i\ s\ s' \neq \star} \pi_2\ i\ s\ s' \right) \leq 1. \quad [18]$$

This ensures the most weight a single identity can cause is 1, no matter how many intents they spam. Such identity content is outside the scope of an intent machine, however.

2.5. Example: Number Machine. To clarify this formulation on an example, let's attempt to create an intent machine where the state is just a number, and the intents describe preferences for the transition between numbers.

For practical purposes, the state will necessarily be finite. We will assume that the number is an element of $\mathbb{N}$, that is, a natural number.

To keep it simple, we will assume we have the same four intents at each step, desiring:

1. The next state should be even.
2. The next state should be odd.
3. The next state should differ by 1 from the last.
4. The next state should be 1 or 2 greater than the last. If 2 greater, full score, if 1 greater, half score, otherwise, no score.

So the input stream is just a set of these 4 intents, repeated forever.

We may formalize this as an ILP problem. To demonstrate this, we will use SCIP through Google OR-Tools.

```
from ortools.linear_solver import pywraplp
```

To make boolean variables correspond to our desired propositions, we will use the "Big M" method. For demo purposes, M does not need to be that large;

```
M = 1000
```

The intents themselves can be formulated as functions that add constraints to the solver. In a more abstract setting, we may think of syntactic encodings of the constraints as R. For demo purposes, the constraints are added to the solver, and it's the booleans linked to the constraints that are returned. The scores are tied to the booleans themselves. We ensure that the total score of the booleans is between 0 and 1. It may not always be obvious how to do this, as some booleans may not be exclusive. For this example, all booleans are exclusive, but, in general, such an encoding may be inefficient. The current state is taken in as a constant argument, while a variable for the next state is taken as an additional argument so all the constraints can talk about the same next state.

```
def even_constraint(last_state, next_state_var, solver):
    n = solver.IntVar(0, solver.infinity(), 'n_even')
    b = solver.BoolVar('b_even')
    solver.Add(next_state_var - 2 * n ⩽ M * (1 - b))
    solver.Add(next_state_var - 2 * n ⩾ -M * (1 - b))
    return [(b, 1)]

def odd_constraint(last_state, next_state_var, solver):
    n = solver.IntVar(0, solver.infinity(), 'n_odd')
    b = solver.BoolVar('b_odd')
    solver.Add(next_state_var - 2 * n - 1 ⩽ M * (1 - b))
    solver.Add(next_state_var - 2 * n - 1 ⩾ -M * (1 - b))
    return [(b, 1)]

def two_changes_constraint(last_state, next_state_var, solver):
    b_2_greater = solver.BoolVar('b_2_greater')
    b_1_greater = solver.BoolVar('b_1_greater')
    b_no_change = solver.BoolVar('b_no_change')

    solver.Add(next_state_var - (last_state + 2) ⩽ M * (1 - b_2_greater))
    solver.Add(next_state_var - (last_state + 2) ⩾ -M * (1 - b_2_greater))
    solver.Add(next_state_var - (last_state + 1) ⩽ M * (1 - b_1_greater))
    solver.Add(next_state_var - (last_state + 1) ⩾ -M * (1 - b_1_greater))
    solver.Add(next_state_var - last_state ⩽ M * (1 - b_no_change))
    solver.Add(next_state_var - last_state ⩾ -M * (1 - b_no_change))

    return [(b_2_greater, 1), (b_1_greater, 0.5), (b_no_change, 0)]

def one_more_or_less_constraint(last_state, next_state_var, solver):
    b1 = solver.BoolVar('b_one_more')
    b2 = solver.BoolVar('b_one_less')

    solver.Add(next_state_var - (last_state + 1) ⩽ M * (1 - b1))
    solver.Add(next_state_var - (last_state + 1) ⩾ -M * (1 - b1))
    solver.Add(next_state_var - (last_state - 1) ⩽ M * (1 - b2))
    solver.Add(next_state_var - (last_state - 1) ⩾ -M * (1 - b2))

    return [(b1, 1), (b2, 1)]
```

These are close to the representation formulation involving R. Each takes, as an initial argument, last_state, and produces a representation of the proposition in the form of a Python function that takes a variable representing the next state and a solver, and it returns a list of variables along with its weight. The analog of R in these functions are the various statements added to solver. Due to the stateful nature of ortools, this cannot be made to look exactly like the ideal formulation, but hopefully, the connection is clear enough.

The machine itself has essentially the same formulation as the theory; being a function taking a state and a batch of intents, represented as a dictionary mapping constraint names to functions and weights. These weights, which do not appear in the abstract intent machine description, are the output of a hypothetical reputation system dictating how important each constraint is. These will all be 1 for this demo. Note that these are unrelated to the weights returned by the intents themselves. These intents are then called to modify the solver state to include the new constraints. The sum of the booleans scaled by weight is then used as an objective function, and a new state, along with a list of satisfied intents is returned. There's also a 30-second timeout, but the demo takes only microseconds.

```
def machine(state, constraints_dict):
    solver = pywraplp.Solver.CreateSolver('SCIP')
    if not solver:
        raise Exception('SCIP solver not available.')

    solver.SetTimeLimit(30000)

    next_state_var = solver.IntVar(0.0, solver.infinity(), 'next_state_var')

    objective = solver.Objective()
    objective.SetMaximization()
```

```
    bool_vars = {}  # Dictionary to store solver boolean variables

    for name, (constraint_func, weight) in constraints_dict.items():
        for b, w in constraint_func(state, next_state_var, solver):
            bool_vars[b] = name  # Store the constraint name
            objective.SetCoefficient(b, w * weight)

    status = solver.Solve()

    total_objective = objective.Value()

    satisfied_constraints = []

    if status in [pywraplp.Solver.OPTIMAL, pywraplp.Solver.FEASIBLE,
    ↪  pywraplp.Solver.ABNORMAL]:
        new_state = next_state_var.solution_value()
        for b in bool_vars.keys():
            if b.solution_value() > 0.5:
                satisfied_constraints.append(bool_vars[b])  # Get the constraint name
        return new_state, satisfied_constraints, total_objective
    else:
        return state, [], total_objective
```

as a usage example, we can put all our constraints into the batch;

```
constraints_dict = {
    'even': (even_constraint, 1),
    'odd': (odd_constraint, 1),
    'two_changes': (two_changes_constraint, 1),
    'one_more_or_less': (one_more_or_less_constraint, 1)
}
```

we can then run

```
new_state, satisfied, total_objective = machine(5.0, constraints_dict)
print(f"New state: {new_state}, Satisfied constraints: {satisfied}, Total Objective:
↪  {total_objective}")
```

getting the output

```
New state: 6.0, Satisfied constraints: ['even', 'two_changes', 'one_more_or_less'],
Total Objective: 2.4999999999999996
```

We may then implement a function to actually run the machine on a stream of batches;

```
def run_machine_in_sequence(initial_state, list_of_constraints_dicts):
    current_state = initial_state
    output_states = []
    all_satisfied_constraints = []

    for constraints_dict in list_of_constraints_dicts:
        new_state, satisfied_constraints, total_objective = machine(current_state,
        ↪  constraints_dict)
        output_states.append(new_state)
        all_satisfied_constraints.append((satisfied_constraints, total_objective))
        current_state = new_state  # Update the current state for the next iteration

    return output_states, all_satisfied_constraints
```

This function will repeatedly stream batches from an input list and update the state accordingly. The history of states and satisfied constraints will then be outputted after the list of batches is exhausted. We can see the state evolves over time with

```
run_machine_in_sequence(0, [constraints_dict for x in range(10)] )
```

This is just going to repeat the batch with all 4 intents 10 times. It will output.

```
([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0],
 [(['odd', 'two_changes', 'one_more_or_less'], 2.5),
  (['even', 'two_changes', 'one_more_or_less'], 2.5),
  (['odd', 'two_changes', 'one_more_or_less'], 2.5),
  (['even', 'two_changes', 'one_more_or_less'], 2.5),
  (['odd', 'two_changes', 'one_more_or_less'], 2.4999999999999996),
  (['even', 'two_changes', 'one_more_or_less'], 2.4999999999999996),
  (['odd', 'two_changes', 'one_more_or_less'], 2.5),
  (['even', 'two_changes', 'one_more_or_less'], 2.5),
  (['odd', 'two_changes', 'one_more_or_less'], 2.4999999999999996),
  (['even', 'two_changes', 'one_more_or_less'], 2.4999999999999996)])
```

It never seeks to satisfy the full "two-greater" constraint. Doing so would only gain 0.5 while losing 1 from not satisfying the one-more-or-less constraints. We can, at this point, create arbitrary streams of batches to get more interesting histories, if desired.

3. Barter Exchange preliminaries

3.1. Introduction. One of the goals of Anoma is to enable *barter*, where different users express their preferences for *resource* exchanges, discover counterparties, and execute these exchanges. We will now analyze how we can instantiate intent machines, or approximations thereof, which fulfill the requirements of a resource bartering system.

Before we introduce definitions of different *bid machines*, we first describe *barter double auctions*[1] for the Anoma Abstract Resource machine, which is a type of *barter exchange*[2] we want to implement.

Specifically in our setting, *agents* express *demands* and *offers* of Anoma *resource* bundles, which are specified by *predicates*, with *solvers* acting as *auctioneers*, matching demand and offer bundles of multiple agents according to a given *mechanism*.

3.2. Resource Model. In practice, the resource system would be used as follows: A *resource kind* would model a *type of object*, instances of which are fungible with each other, e.g. identical teapots. Each *resource* would indicate a *specific* teapot and its properties, e.g. which user in the system currently owns it. When the teapot is transferred, the resource representing it is consumed, and a new resource, representing the same teapot indicating the new owner, is created.

Let us now introduce a simplified version of the Anoma resource model[3] to lay out the objects of interest in the barter. This introduction is reduced to the features relevant to the characterization we wish to analyze here.

- A set of *Agents*, which are entities that can instantiate data structures, have access to secret information that enables them to create specific signatures $sig \in \mathbb{S}$ and can own resources (see below). In general, no agent has access to information that would enable them to generate all potential signatures.
- The *set of all possible resources* $\mathcal{R}$ is induced by a specific hash function h, containing one resource per element of the image of h.
- *Resources* are the elements of the state $\mathcal{S}$ on which a bid machine operates. It consists of the set of created resources $\mathcal{S}_+ \subseteq \mathcal{R}$ and the set of consumed resources $\mathcal{S}_- \subseteq \mathcal{R}$ with $\mathcal{S}_+ \cap \mathcal{S}_- = \varnothing$ and $\mathcal{S} = (\mathcal{S}_+, \mathcal{S}_-)$.
- An Agent can add a to $\mathcal{S}_+$ by *creating* it, via instantiating it as a data structure which fulfills the predicate from the resource kind (see below). One of these requirements could include consuming a resource of the same kind.
- When a resource is *consumed* (which must only happen when the requirements of its kind are fulfilled), it moves from $\mathcal{S}_+$ to $\mathcal{S}_-$. Resources can only be consumed once, and they cannot move back to $\mathcal{S}_+$.
- A *transaction* implements a state transition by consuming and creating sets of resources $TX : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$[4]. $TX \in \mathcal{T}$, which is the set of all possible TXs over $\mathcal{S}$.
- Resources $r = (k, v) \in \mathcal{R}$ have the following relevant properties:
  - A *kind* k, which determines restrictions on TXs this resource can be part of, e.g. mandating which resources can and need to be consumed or created in the TX. It can also require signatures $\in \mathbb{S}$, which depend on secret information and is encoded in a predicate; $k : \mathcal{T} \times \mathcal{P}(\mathbb{S}) \to \mathbb{B}$.
  - A *value* v, which can contain arbitrary application data (e.g. a public key of a user controlling a particular resource).

[1] Tagiew (2009)
[2] For more background on (robust) barter exchange, see chapter 6 of Glorie (2014)
[3] Khalniyazova and Goes (2024)
[4] $\mathcal{P}(X)$ denotes the powerset of a set X

- *Ownership* of a resource means having the capabilities to fulfill its k. Ownership can be transferred by the current owner fulfilling k to enable the consumption of a resource, while in the same TX creating a new resource which carries a k that enables only the new owner to fulfill it. This can be achieved by making k depend on, e.g. a public key encoded in v that only the new owner can produce a signature for.

3.3. Bids. To be able to describe barter exchanges over resources, we need to formulate *barter intents*, which are usually called *bid* in the auction literature. In a case where a single good is up for auction in exchange for money (e.g. a sealed-bid second-price auction) they indicate what the maximum price a party is willing to pay to acquire the good. This generalizes to vector valued bids in cases where multiple items are up for auction.

For the case of two sided barter auctions, which we want to enable, we need to generalize further. Since the items to be auctioned are aggregated from the bids the participants submit, the simplest possible bid type would consist of an offer vector, indicating how many of which item type an agent is putting into the pool, and a demand vector, indicating how many of which item types they want to receive in return. This would still be of limited expressivity, since there is no way to rank preferences or to state alternative, but mutually exclusive bids. Thus we introduce bids using predicates to indicate whether a demand is met and what shapes the offer could take in this case.

A bid, as used in our model, is defined as follows:

$$\begin{aligned} bid &= (p, u, rs) \\ p &: \mathcal{T} \to \mathbb{B} \\ u &: \mathcal{T} \to [0, 1] \subset \mathbb{Q} \end{aligned} \tag{19}$$

Predicates p range over TXs and return `true` if a TX is acceptable to the issuer. Internally, they specify the offer and demand sides. Since p can accept different TXs as valid, a utility function u is used to order them by preference. A set of bids $\{bid_1, \ldots, bid_n\}$ is *matched* if a TX can be composed from the resources of their offer sides (see below), such that all p_i, as well as all k from the resources r_{TX} used in the TX ($r_{TX} \subseteq \bigcup_{i=1}^{n} \text{offer}(p_i)$) are satisfied. Given there are no alternatives, the ranking induced by u is irrelevant.

- The *offer* side of a predicate[5] (offer(p)) specifies which resources the issuer owns and could give up for consumption in a TX. **It is required that each resource in the offer side is owned by the issuer.**
- The *demand* side of a predicate (demand(p)) specifies which resources need to be created in a TX, with the issuer of the bid as owner.

Let us look at an example barter[6], with Agent 1 issuing bid_1:

- `p: (Offer: 1 Bread OR 1 Teapot) AND (Demand: 1 Pen)`
- `u: (Give: 1 Bread) AND (Get: 1 Pen): 1.0, (Give: 1 Teapot) AND (Get: 1 Pen): 0.8, (Give: 1 Bread, 1 Teapot) AND (Get: 1 Pen): 0.5`

And Agent 2 issuing bid_2:

- `p: (Offer: 1 Pen XOR 1 Teacup) AND (Demand: 1 Teapot AND 1 Bread)`
- `u: (Give: 1 Pen) AND (Get: 1 Teapot, 1 Bread): 1.0, (Give: 1 Teacup) AND (Get: 1 Teapot, 1 Bread): 0.1`

A TX that would satisfy both predicates using only resources from $offer(p_1) \cup offer(p_2)$, with Agent 1 receiving 0.5 utility and Agent 2 1.0, could like this:

- `Consume: {Agent 1: 1 Bread AND 1 Teapot, Agent 2: 1 Pen}`
- `Create: {Agent 1: 1 Pen, Agent 2: 1 Bread, 1 Teapot}`

3.4. Bids implemented by transactions. Every bid, p, u, rs implies a set of potential TXs, since only the offer side is given. A set of matching bids, b_i, is where the offer and demand sides of every b_i are met, which can be directly derived from the bid set. Any TX that contains independent subsets of matching bids can be decomposed into smaller TXs corresponding to the independent subsets of matching bids. These smaller independent transactions can all be executed in parallel. We call a set of TXs derived from a set of matching bids a *barter exchange* BX.

[5] In practice, offer and demand side can be interleaved in a predicate, but for simplicity, let us assume they are separate structurally.
[6] We use monospace font here to make comparison of bids easier.

3.5. Bid ⇔ Intent correspondence. The above is compatible with our definition for Intents I, [12], as follows:

- p implies a set of accepted state transitions, each element of which can be expressed as: $t : S \to S := s \mapsto s'$, with $S = \mathcal{S}$.
- u gives a weighting $w(x) = u(x)$ for the state transitions t, not rejected by p. $w : S \times S \to [0, 1]$

Putting this all together, we get:

$$\begin{aligned} bid &= (t, \lambda s.\ \text{if } p(s) \text{ then } u(s) \text{ else } \star), \\ bid &: (\mathcal{S} \to \mathcal{S}) \times (\mathcal{S} \times \mathcal{S} \to \star \cup [0, 1]) \end{aligned} \tag{20}$$

3.5.1. Bid examples. The predicates on both sides can be used to specify offer/demand pairs of arbitrary bundles of resources, for example:

- Single resources on both sides
- Dependent bundles, e.g. if an agent wants a resource of kind A only if they can acquire one of kind B or C at the same time

The predicate specifies a set of accepted state transitions, with the utility function ranking them. Outside of satisfying the predicates, the auction mechanism M executed by the auctioneer can be chosen freely.

3.5.2. Bid composition. Multiple bids can be composed into a single bid, for example like this:

$$bid_{1 \wedge 2} = ((p_1 \wedge p_2), (u_1 + u_2)) \tag{21}$$

With $u_i + u_j := \lambda x. u_i(x) + u_j(x)$. More complex dependencies of bids can be expressed via predicates, e.g. "The offer from bid_1 is only valid if the demand from bid_2 is fulfilled in the same TX.

3.6. Roles. We also have the following roles:

- A group of *Agents*, which submit bids $b = \{bid_1, \ldots, bid_n\}$ to an auctioneer. These are the Agents from the resource model.
- One or several *Auctioneers*[7], which receive the bids and match subsets of these according to offer and demand sides from their predicates. The matching maximizes a metric given by the auctioneer. From this, we derive a barter exchange BX and submit it to the controller for ordering. The auctioneer can submit BX containing a single TX as soon as a matching set of bids is found, or wait until it has processed more or all of the batch of bids, submitting a larger BX.
- One *Controller*, which manages state and orders TX candidates, contained in the BXs, before the executor reads them. Time is sliced by discrete *ticks*, during which TX candidates are arranged in a partial order, by order of submission to the controller, depending on some ordering mechanism[8]. The end of the current tick and the beginning of the next is delineated by the executor performing a state update.
- One *Executor*, which executes candidate TXs. This is done by applying the state updates contained in the TX candidates which are valid at execution time, as well as computing and publishing a cumulative state update. Details of this behavior are determined by the execution mechanism E.

4. Bid Machines

Let us now introduce two types of bid machines: a restricted one, which instantiates an intent machine directly, and a more general one, which does not. The case distinction we need to make depends on when and where the auctions are computed.

We have the following objects:

- s_0, the state of the underlying resource machine prior to the auction
- b_0, a batch of bids by the agents
- M, the concrete auction mechanism which determines the selection of b_1
- b_1, the subset of bids chosen by an auctioneer
- BX a set of TXs derived from b_1

[7] these auctioneers act as solvers
[8] The ordering mechanism is out of scope for this report.

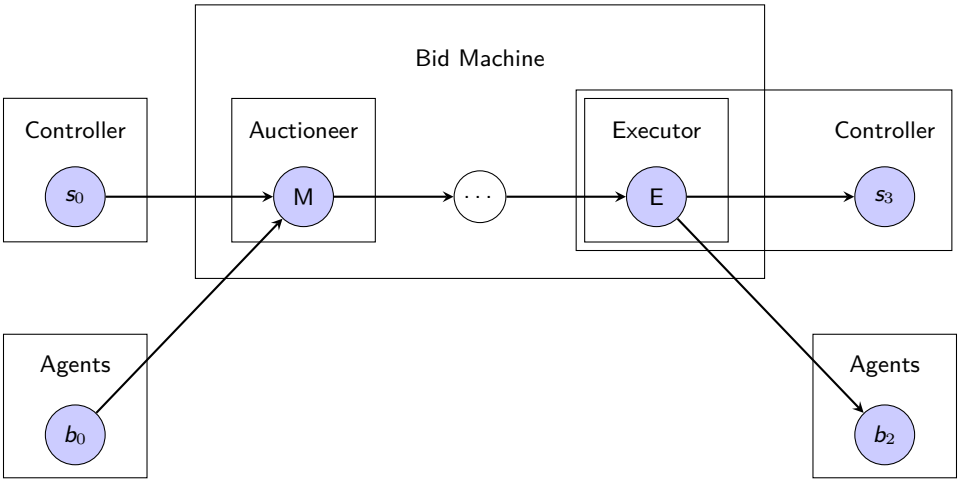

Figure 1. External bid machine interface, internal decomposition details omitted.

- s_1, the state after auction computation (only in controller-dependent case, see below)
- s_2, the state prior to execution
- E, the execution mechanism, which performs state updates, by executing TXs from BXs it receives
- s_3, the new state after s_2 was updated by E
- b_2, the subset of b_1 which was not taken into account when updating to the new state s_3, i.e. the bids contained in failed TXs, as well as unmatched ones

The **bid machine is defined by *M* and *E***, both of which are determined by decisions which happen out of band of this abstraction level.

M implements a bartering auction over bids, its specific parameters determining its properties, e.g., regarding robustness or the choice of metric to maximize. It can also depend on reputation-tracking relationships between agents, auctioneers, and controllers. M is only responsible for matching the bids, but during the derivation of a BX from b_1, certain choices can be made too, since in general, there is no unique mapping. The details of this are out of scope for this report.

E implements an execution mechanism for barter exchanges, it is uniquely defined.

Note: The concrete types of M and E differ in the on-controller/controller-dependent and off-controller/controller-independent settings, regarding the details how the bid matchings are propagated through the system, i.e. via an embedding in the state or passing batches.

4.1. Auction computed on the controller. Let us start with a restricted setting, in which the auction is computed on the controller, which means the auctioneer and executor have access to the current state of the system at all times.

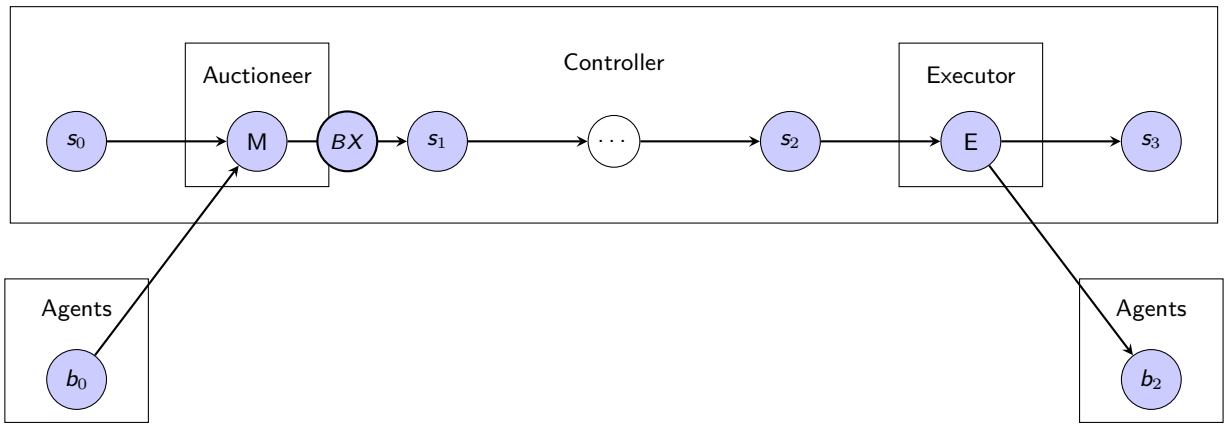

Figure 2. Information flow in a controller-dependent Bid Machine.

In this setting, M and E each instantiate an intent machine on their own:

$$M(s_0, b_0) = (s_1, \varnothing)$$
$$E(s_2, \varnothing) = (s_3, b_2) \quad [22]$$

4.1.1. Controller dependent Bid Machine ⇔ Intent Machine correspondence. To instantiate an intent machine according to our definition in [1], the composition needs to happen in the following way:

Auctioneer:

- reads b_0
- reads s_0
- computes matching b_1, according to M
- writes BX derived from b_1 to s_0, not touching any other part of s_0, receiving s_1 as a result
- outputs s_1 as computed above, and an empty batch

As we can see, we do not output b_1 or BX, but only s_1 in which BX is embedded.

Executor:

- ignores its input batch
- reads the state s_2 which includes BX (see below for cases $s_1 = s_2, s_1 \neq s_2$)
- deletes BX from the state
- executes according to E, updating the state to s_3
- outputs b_2, containing all bids which were not matched

Controller:

- only a single one exists
- implements the roles of auctioneer and executor
- If the computation of M and execution with E happen during the same tick, with M being computed at post-ordering time, then $s_1 = s_2$. This is the recommended setup. If M is computed a previous tick, state might have changed for reasons outside of this bid machine between s_1 and s_2.

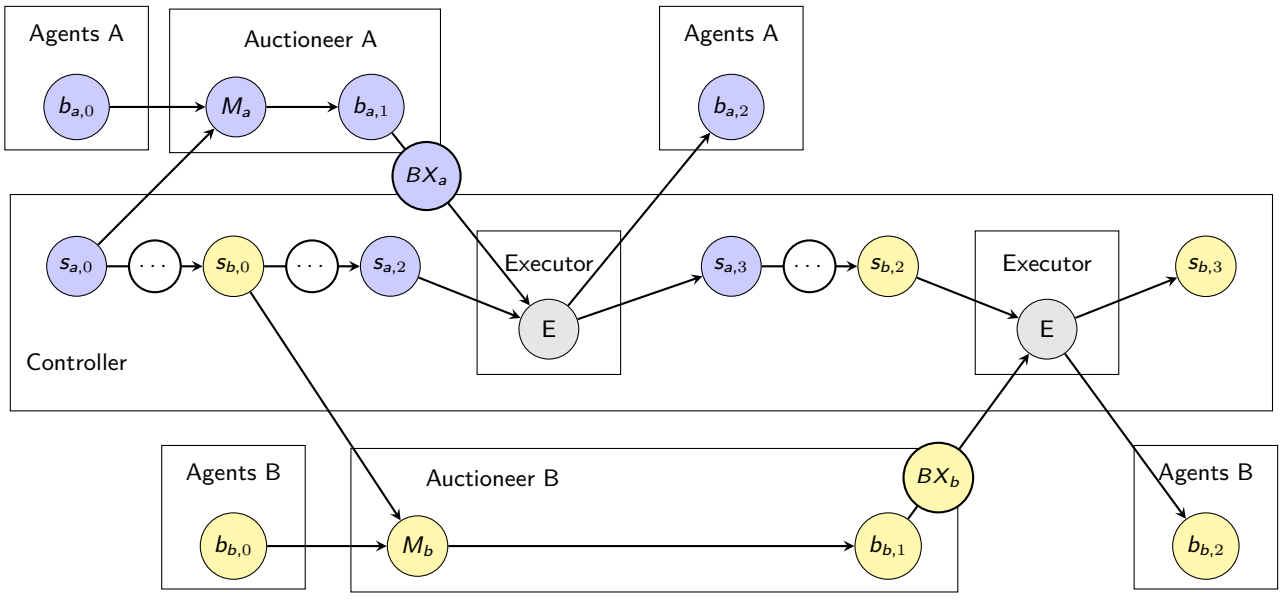

Figure 3. Information flow of interleaved controller independent Bid Machines.

4.2. Auctions computed independently of the controller. In the given context, none of the entities M, E, or any individual bid machine instantiate an intent machine. Instead, the intent machine is instantiated by all bid machines on the same controller, if any of them operate in this setting.

$$M(s_0, b_0) = (\varnothing, b_1)$$
$$E(s_2, b_1) = (s_3, b_2) \quad [23]$$

In Fig. 3, we can see two off-controller auctions

- A (blue objects) with $\{b_{a,0}, b_{a,1}, b_{a,2}, M_a, s_{a,0}, s_{a,1}, s_{a,2}, s_{a,3}\}$
- B (yellow objects) with $\{b_{b,0}, b_{b,1}, b_{b,2}, M_b, s_{b,0}, s_{b,1}, s_{b,2}, s_{b,3}\}$

sharing the same, single executor and E, which runs on a single controller[9].

Auctioneers:

- read $b_{a,0}, b_{b,0}$
- read $s_{a,0}, s_{b,0}$
- compute matching $b_{a,1}, b_{b,1}$, according to M_a, M_b
- output BX_a, BX_b derived from $b_{a,1}$ or $b_{b,1}$ respectively

Executor:

- reads BX_a, BX_b
- reads the state $s_{a,2}, s_{b,2}$
- executes BX_a, BX_b, according to E updating the state to $s_{a,3}, s_{b,3}$
- outputs $b_{a,2}, b_{b,2}$, containing all bids which where not included in execution and $s_{a,3}, s_{b,3}$

Controller:

- only a single one exists
- implements the only executor
- orders any TXs contained in BX_a, BX_b
- at all points marked with "...", arbitrary things can happen to the state due to other transactions being executed
- Interleaved operation is the most general case, but if $s_{a,2} = s_{b,2}$ when $b_{a,0}$ and $b_{b,0}$ are submitted, both auctions happen during the same tick. Then TXs from both BX_a and BX_b are taken into account when ordering TXs. After execution $s_{a,3} = s_{b,3}$.

4.2.1. Controller independent Bid Machines ⇔ Intent Machine correspondence. This generalizes to arbitrary numbers of agent sets and auctioneers, indexed by $i \in \mathbb{N}$, sets of bids $\mathcal{B} = \{b_{a,0}, \ldots, b_{n,0}\}$ and states $\mathcal{S}_0 = \{s_{1,0}, \ldots, s_{n,0}\}$, with $\mathcal{B}_1, \mathcal{B}_2, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ analogously. We assume a total temporal ordering of all elements (produced by the executor) of the state sets, but their indices are independent of their position in the order, instead denoting which bid machine they belong to. In general, agent sets do not have to be disjoint.

This setup only fulfills our definition of intent machine if we take:

$$\begin{aligned} b_0 &= b_{1,0} \cup \cdots \cup b_{n,0} \\ b_2 &= b_{1,2} \cup \cdots \cup b_{n,2} \\ s &= \min \mathcal{S} \quad \text{where min is first state in the set} \\ s_3 &= \max \mathcal{S}_3 \quad \text{where max is last state in the set} \\ E \circ M(s_0, b_0) &= (s_3, b_2) \end{aligned} \quad [24]$$

As such, **it does not enable any internal decomposition into intent machines**. A refined notion that enables decomposition of this general setting will be the subject of a subsequent publication.

4.3. Composition of Bid Machines. There are two different ways to compose auctions in our model.

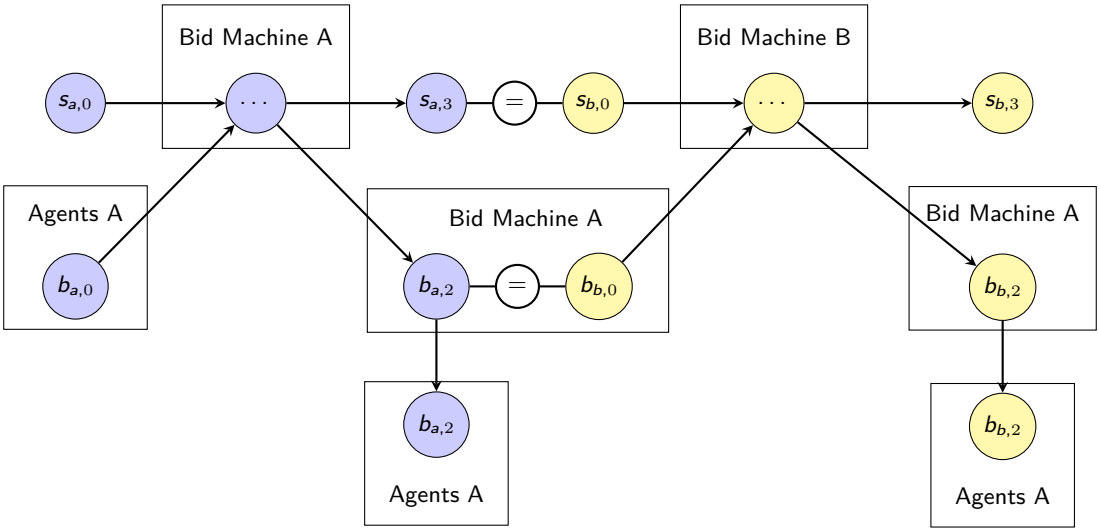

Figure 4. Sequential composition of two barter exchanges.

4.3.1. Sequential composition. The first bid machine reads $s_{a,0}, b_{a,0}$, internally computes a matching with M_a and executes with E_a, returning $s_{a,3}$ and $b_{a,2}$. In turn, $s_{a,3}$ and $b_{a,2}$ get fed into the next bid machine with $s_{b,0} = s_{a,3}$ and $b_{b,0} \subseteq b_{a,2}$. In general, $s_{b,0}$ could also happen at any point after $s_{a,3}$ is returned, with arbitrary state changes happening in between.

Here we do not care about whether the auctions are computed in the controller dependent (in which M writes state for E to read at "..." time), or independent setting (in which M passes b_1 directly to E), since the only relevant objects being passed around are state s_0 and bids b_0, and only before the start of an auction and after the end of execution.

Choosing the bids which auctioneers hand on is up to them. The choice of internal mechanisms M_i and their behavior under composition will inform their strategies. We will elaborate on these game theory and mechanism design questions in further publications.

If auctioneers collate bids from different sets of agents before forwarding them, any b_2 need to be split up and reported to the correct agents in the end.

4.3.2. Parallel composition. Parallel composition means running the bid machine on a union (of subsets of) the bids received by bid machines during one tick of a given controller. For this, we need to do parallel composition of the auctioneers and executors individually. Composing executors in parallel means all executors are collapsed into a single one[10]

For auctioneers we need to characterize a spectrum of cooperation with three axes, the specific parameters of which are decided upon by external governance.

The first axis is **bid disclosure**, with the following bounding cases:

- Full mutual bid disclosure: All auctioneers disclose each other all bids received during the current tick.
- No mutual disclosure.

The second axis is **selection of bids** to search over, with the following bounding cases:

- Full selection cooperation: All auctioneers coordinate the selection of bids to search over.[11]
- No selection cooperation: Every auctioneer selects bids independently.

The third axis is the **choice of mechanism** M, which is applied to the set of bids, to search for BXs:

- Full mechanism cooperation: All auctioneers coordinate on the mechanism(s) used.[12]
- No selection cooperation: Every auctioneer selects bids independently.

For simplicity in Figure 5, we assume $b_{a,0}, b_{b,0}$, have been read from the agents in the beginning and $b_{a,2}, b_{b,2}$ are reported to the agents in the end. Further, let $\dot{b}_i \subseteq b_i$ be the bids an agent shares with another.

[9] The roles of Agents A+B, as well as Auctioneers A+B *could* in practice be implemented by the same entities.

[10] This can happen by every executor delegating to the same executor. Or, since in practice, controllers and executors will often be implemented by consensus providers, the collapsing could be done e.g. via spinning up a chimera chain, see https://anoma.net/blog/chimera-chains.

[11] This does not necessarily mean agreeing on a single set, to search together. They could, e.g. agree on a partition and search individually, but the fact that it is coordinated is relevant.

[12] They could either pick one mechanism and trust one auctioneer to execute it, potentially verifying the result. Other options are to execute a single mechanism in some distributed fashion, or coordinate on mechanisms each party independently executes.

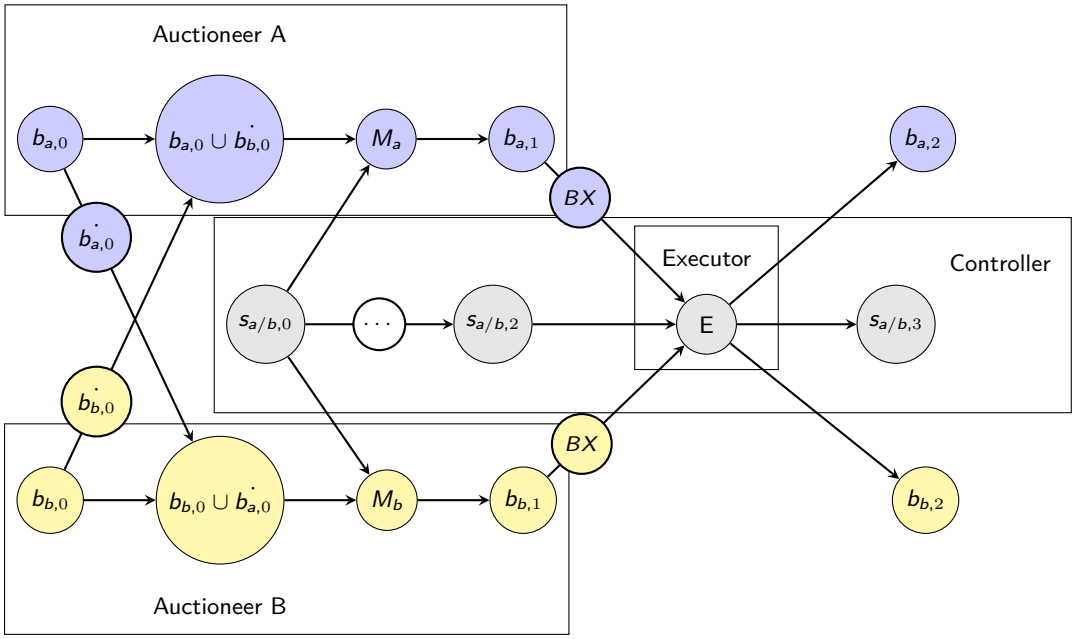

Figure 5. Information for bid sharing in parallel composition.

The following holds:

$$\begin{gathered} \dot{b_{a,0}} = b_{a,0} \wedge \dot{b_{b,0}} = b_{b,0} \Rightarrow b_{a,0} \cup \dot{b_{b,0}} = b_{b,0} \cup \dot{b_{a,0}} \\ \text{if the above and } M_a = M_b \Rightarrow b_{a,1} = b_{b,1} \wedge b_{a,2} = b_{b,2} \end{gathered} \quad [25]$$

How computational cooperation will function in detail will be the subject of another publication. Broadly, the choices range from trusting a single party with computation, potentially verifying the correctness of the results where possible, or running a distributed computation between (a subset of) the auctioneers.

To improve the guarantees for bid sharing, instead of sending them directly to the auctioneers, they could be ordered by the controller, adding a side constraint to all bids that any resource from R_o can only be used if controller signatures are present. This would give guarantees against insertion or censorship of bids submitted to the verified set by the auctioneers during a single round. This can be done in the on-controller and off-controller auction settings.

5. Conclusion and Future Work

We have introduced intent machines, formulated them in the abstract using coalgebras, defined their most general notions of composition, demonstrated a toy example, and defined notions of such machines for barter. These efforts seek to define a minimal formulation as a basis for characterizing the game theoretic model, especially informing further mechanism design research.

6. Acknowledgements

We would like to thank Isaac Sheff for valuable discussions and feedback on the implementation of controllers and details of the distributed state machine, Yulia Khalniyazova for feedback on both reading experience and finding a simplified but still correct model of the Anoma Resource Machine, Artem Gureev for making sure the abstract notions make sense through formalization efforts, Christopher Goes for helping clarify how the different components integrate across different levels of abstraction, and Tobias Heindel, Jonathan Prieto-Cubides and Isaac Sheff for copy-editing assistance.

References

Bart Jacobs. *Introduction to coalgebra*, volume 59. Cambridge University Press, 2017. (cit. on p. 2.)

Benjamin C. Pierce. *Types and Programming Languages*, chapter 23: Universal Types. MIT Press, Cambridge, MA, USA, 2002. (cit. on p. 3.)

Marcello M Bonsangue, Jan Rutten, and Alexandra Silva. Coalgebraic logic and synthesis of mealy machines. In *International Conference on Foundations of Software Science and Computational Structures*, pages 231–245. Springer, 2008. (cit. on p. 3.)

Clemens Kupke and Dirk Pattinson. Coalgebraic semantics of modal logics: An overview. *Theoretical Computer Science*, 412(38):5070–5094, 2011. (cit. on p. 4.)

Rustam Tagiew. Towards barter double auction as model for bilateral social cooperations, 2009. (cit. on p. 9.)
Kristiaan Glorie. *Clearing Barter Exchange Markets: Kidney Exchange and Beyond*. PhD thesis, E, November 2014. URL http://hdl.handle.net/1765/77183. (cit. on p. 9.)
Yulia Khalniyazova and Christopher Goes. Anoma resource machine specification, January 2024. URL https://doi.org/10.5281/zenodo.10498991. (cit. on p. 9.)

7. Appendix

7.1. Proof that $\mathcal{M}$ is a monad. It may not be obvious that $\mathcal{M}$ is, indeed, a monad. To prove this, we must define a notion of monad return and monad bind satisfying the monad laws,

- return $a >>= h \equiv h\ a$
- $m >>=$ return $\equiv m$
- $(m >>= g) >>= h \equiv m >>= (\lambda x.g\ x >>= h)$

We are forced by the types of the operators to have the following implementations;

- return $a := \lambda b.\text{return}_M\ (a, b)$
- $a >>= f := \lambda b.a\ b >>=_M (\lambda(x, b).f\ x\ b)$

With these definitions, we can verify the laws with equatorial reasoning.

$$\begin{aligned}
\text{return } a >>= h &= \lambda b.(\lambda b.\text{return}_M(a, b))b >>=_M \lambda(x, b).h\ x\ b \\
&= \lambda b.\text{return}_M(a, b) >>=_M \lambda(x, b).h\ x\ b \\
&= \lambda b.(\lambda(x, b).h\ x\ b)(a, b) \\
&= \lambda b.h\ a\ b \\
&= h\ a
\end{aligned} \tag{26}$$

$$\begin{aligned}
m >>= \text{return} &= \lambda b.mb >>=_M \lambda(x, b).(\lambda a\ b.\text{return}_M(a, b))\ x\ b \\
&= \lambda b.m\ b >>=_M \lambda(x, b).\text{return}_M(x, b) \\
&= \lambda b.m\ b >>=_M \text{return}_M \\
&= \lambda b.m\ b \\
&= m
\end{aligned} \tag{27}$$

$$\begin{aligned}
(m >>= g) >>= h &= \lambda b.(\lambda b.m\ b >>=_M (\lambda(x, b).g\ x\ b))\ b >>=_M \lambda(x, b).h\ x\ b \\
&= \lambda b.(m\ b >>=_M (\lambda(x, b).g\ x\ b)) >>=_M \lambda(x, b).h\ x\ b \\
&= \lambda b.m\ b >>=_M (\lambda(x, b).(\lambda(x, b).g\ x\ b)(x, b) >>=_M \lambda(x, b).h\ x\ b) \\
&= \lambda b.m\ b >>=_M (\lambda(x, b).(g\ x\ b >>=_M \lambda(x, b).h\ x\ b) \\
&= \lambda b.m\ b >>=_M (\lambda(x, b).(\lambda b.g\ x\ b >>=_M \lambda(x, b).h\ x\ b)\ b) \\
&= \lambda b.m\ b >>=_M (\lambda(x, b).(\lambda x\ b.g\ x\ b >>=_M \lambda(x, b).h\ x\ b)\ x\ b) \\
&= m >>= (\lambda x.g\ x >>= h)
\end{aligned} \tag{28}$$

This proves that $\mathcal{M}$ is, indeed, a monad.