Draft Proposal: Intents as Constraint Satisfaction Problems, implemented by (Partial) Transactions

Co-authored by: @AHart

Thanks for feedback to: @vveiln , @cwgoes

It seems to us that one natural formulation of Solving for Intents

is in the form of Constraint Satisfaction Problems. What we are trying to achieve is performing counterparty discovery for the purpose of optimizing preference satisfaction with respect to which parts of the state space an agent wants to give up control over, and for which they would like to receive control in exchange for it, i.e. finding good matches under constraints.

The following is a draft proposal on this formulation. A lot of details still need to be worked out, we are keen to receive any kind of feedback, suggestions and comments and will iterate accordingly.

Brief introduction to the Anoma Resource Model

Since we want to bridge the gap between matching points in a state space to CSP instance formulations, let us first look at the implementation of the state space we're working with:

In anoma, state is modeled using immutable Resources

. For the sake of explanation, we will introduce a simplified version of it here (i.e. omitting cryptographic details).

struct Resource { res_type: String, predicate: fn PTX -> bool, // Determines which conditions need to be fulfilled for a Resource to be consumed or created nonce: ByteString, owner: Identity, quantity: Int, value: ByteString, ephemeral: bool, }

The Resource Type, res_type

indicates a name space, tied to a Predicate

governing the creation and consumption of Resources

within the name space. A nonce

is used to determine which specific immutable

Resource

from this namespace is being referred to. Any specific Resource

can be consumed exactly once.

Any agent that has access to all the information (e.g. cryptographic keys to authenticate consumption as required by the Predicate

) needed to create a Resource

can do it. Its acceptance by other parties depends on validation by consensus providers they use for reference.

Intents to modify the state are implemented with PTXs

(Partial Transactions).

// Partial Transactions struct PTX { input_resources: Set, // Resources that can be consumed output_resources: Set, // Resources that should be created welfare_function: fn PTX -> float, // welfare_function encodes preferences about search term: fn PTX -> PTX, // term encodes an executable request to different parties (e.g. solvers, consensus providers) on how to transform the PTX, given specified conditions }

PTX

s do not need to be balanced, are created by agents, and can be modified by any computational party (within the bounds of the Resource Predicates

contained within them). Since PTXs

are technically immutable, "modifying" one works as follows in practice: A new PTX

which can include some or all of the Resources from the old one, as well as new Resources

, is created and the old one is discarded.

PTXs

can be submitted to solvers for counterparty discovery, which construct sets that contain balanced input and output Resources

and implement valid state updates, using TXs

(Transactions):

struct TX { ptxs: Set, } // input_ and output_resources respectively of a TX are the aggregates of input_ and output_resources of its constituent PTXs // A TX is always balanced, e.g. for each resource_type, input and output amounts are equal.

TXs

then get sent to consensus providers for ordering. If all input Resources

in a TX

are still available for consumption after ordering, the respective consensus provider signs the TX

as valid and updates its state with the consumed and created Resources

.

Ephemeral Resources

have a lifetime of one TX

and can be created by agents to encode side information for e.g. solvers.

Example: Simple Resource

transfer

agent1 wants to transmit ownership of a Resource

of Resource Type

“Example” to agent2.

let consume = struct Resource { res_type: "Example", predicate: / *consumption authentication, details omitted*/, nonce: 1, owner: agent1, quantity: 1, value: 0x68656c6c6f2c20776f726c6421, ephemeral: false, }

let create = struct Resource { res_type: "Example", predicate: / *consumption authentication, details omitted*/, nonce: 2, owner: agent2, quantity: 1, value: 0x68656c6c6f2c20776f726c6421, ephemeral: false, }

let ptx = PTX { input_resources: [consume], output_resource: [create], welfare_function: none, term: none, }

let tx = TX { ptxs: [ptx], }

Constraint Satisfaction Problems (CSP)

One way to formulate an Intent

over a state transition in the above state space, is as an instance of aConstraint Satisfaction Problem. It is a very natural formulation since we can encode Constraints

over Variables

as PTX

s containing Resources

. This approach enables us to use results from a rich research history and it has interesting special cases, e.g. Linear Programming.

Definition

A CSP is, conceptually, a collection of relations, typically over a single domain, d

. A common example of a CSP is linear programming, consisting of relations such as $x + y \leq z$

and $x = y$

. In that case, the domain is the set of rationals.

Given a CSP defined as a collection of relations, an instance of this CSP consists of

- X

: a set of variables

- C

: a set of constraints consisting of expressions of the form $R(X_1, X_2, ..., X_n)$

, where R

is an n

-ary relation from our CSP, and $X_i \in X$

, for all i

.

An assignment of an instance consists of a map from X

onto d

, and the truth value (or valuation) of an instance relative to an assignment is the evaluation of the conjunction of all constraints with respect to the assignments. If an assignment produces a true valuation, then the instance is said to be “satisfiable”, and the assignment is called a “satisfying assignment” for that instance.

Multi-domain CSPs

In our application, we are interested in multi-domain CSPs. An example of this is mixed-integer linear programming, which has variables ranging over integers, rationals, or booleans. A multi-domain CSP will, instead of having a single domain, instead have a set of domain sets, D

. Additionally, for each n

-ary relation, R

, we will have a tuple $T^R \in D^n$

indicating the domain of each argument.

Instances for multi-domain CSPs can be defined as

- X

: a set of variables.

- d

: a function mapping X

onto D

.

- C

: a set of constraints consisting of expressions of the form $R(X_1, X_2, ..., X_n)$

, where R

is an n

-ary relation from our CSP, and X_i \in X

, for all i

, such that d(X_i) = T^R_i

, for all i

.

Assignments can be extended to this domain in a natural way, as a map from x \in X

onto d(x)

.

It’s worth noting that multi-domain CSPs can be seen as special cases of single-domain CSPs. The single domain CSP will have \cup D

as its domain. We can forget T^R

and treat each relation from the multi-domain CSP as a relation over \cup D

. If need be, we can add predicates to the CSP indicating which set of D

the argument initially came from, allowing one to restrict the domain of variables, though this is usually unnecessary as domains will already be restricted by the relations those variables appear in. This construction illustrates that we don’t gain expressiveness when moving from single to multi-domain CSPs; although keeping domains separated may still be pragmatic for the purposes of implementation.

Correspondence to Transactions

Partial Transactions

will give rise to a variety of constraints depending on their content. The exact CSP instance generated will depend on whether the PTX

is appearing in a full Transaction

or being sent to the solver.

Prior to the creation of a full Transaction

, we will have a floating list of intents. In general, we will not be able to satisfy all intents, so we instead will try maximizing intent satisfaction according to some optimization criteria decided outside of the solver.

As a basic example, if we have a Partial Transaction

, let’s say with index 1

, where some agent wants 5 A

s in exchange for at most 8 B

s, we will have three variables.

- SAT_1

: A boolean indicating if intent 1 is satisfied

- GOT_{1,A}

: An integer indicating how many A

s were received to satisfy intent 1.

- GAVE_{1,B}

: An integer indicating how many B

s were sent to satisfy intent 1.

subject to the constraints

- $GOT_{1,A} = 5 \times SAT_1$
- $0 \leq GAVE_{1,B} \leq 8 \times SAT_1$

These constraints state that the number of A

s received could be either 0

or 5

, depending on if the intent was actually satisfied. Additionally, the number of B

s lost could be between 0

and 8

, so long as the intent is actually satisfied. If the intent is not satisfied, then $GAVE_{1,B}$

will be between 0

and 0

; that is to say, 0

.

Partial Transactions

can also express more complex desires, for example, desiring to trade 2 A

s or 3 B

s, but not both. Such things would require more complex constraints, the details of which may vary. For this example, we may implement it by introducing a boolean variable indicating which resource we are trading, or using a one-hot bit vector instead. Complex constraints can often be implemented in different ways.

Given a collection of Partial Transactions

, we will get a collection of constraints along these lines. To make the whole problem coherent, we'll have additional constraints ensuring the full Transaction

is balanced. For each resource, X

, there will be a constraint that states

$\sum_i GOT_{i,X} = \sum_i GAVE_{i,X}$

where i

ranges over all the indices of all the intents.

The quantity we're trying to maximize could vary, but a basic example might be to try maximizing the number of satisfied intents. In this case, we'd be trying to maximize the quantity;

$\sum_i SAT_{i}$

We may have weights, W_i

, assigned to each intent. We could use these to weigh the importance of certain transactions, transforming our quantity into

$\sum_i W_i SAT_{i}$

Some transactions will also have preferences. An agent can create Ephemeral Resources

carrying predicates that describe properties of the desired output Resources

. Some preferences will turn into constraints (e.g. "I want X

s, but not if they come from agent 47

") while others would influence the optimization criteria (e.g. "I'd rather trade A

s than B

s, but I'll trade both if I have to"). The latter would be specified by a Welfare Function

mapping Partial Transactions

onto a scalar indicating desirability to the agent. The exact way these will be defined is currently up for debate.

As such, the full CSP we are working over is only partially defined, and clarifying its full contents is an active research goal.

A Transaction

containing a set of Partial Transactions

will also be a CSP instance, but one we know will be solvable. Given the output of the constraint solver, we can filter out all mentions of the unsatisfied intents, and simplify out any mentions of SAT_i

(e.g. 0 \leq GAVE_{1,B} \leq 8 \times SAT_1

would turn into 0 \leq GAVE_{1,B} \leq 8

), leaving us with a complete account of Partial Transactions

that are actually doable, according to the solver. The assignment found by the solver will be a satisfying assignment for this filtered CSP instance. This can then be checked by later systems to validate the Transaction

.

Multi Agent, Multi Resource Swaps with Intents

Remarks on this Exposition

In practice, Resource Types

will be commitments of the form hash(resource_predicate, resource_label)

but here we use plaintext fields that describe semantics. We also omit data fields of Resource

s and PTX

s in the service of clarity when they are not required for the explanation. We will use rust syntax, mixed with pseudocode in block comments (/ /).

We omit details of validation (e.g. cryptographic authentication via Ephemeral Resources

) of Resource

consumption and creation since they would obfuscate the CSP model we want to focus on here. A later post will elaborate on this and how it composes. This omission is possible since validation never happens during solving time. Before solving it is used to exclude PTX

s which are invalid. After solving it is used to drop TX

candidates which are invalid where this can only be decided after matching.

Some Resource Predicates

are relevant to validation and some are relevant to solving. In practice, we will express the boundary by using different subsets of the predicate language, which are recognizable at the time of parsing

, s.t. different parts of the system can verifiably distinguish which one are relevant for the current step in the transaction lifecycle.

Example: Multi Agent, Multi Resource swap with fixed Resources and one Linear Preference

Let us now move to a Multi Agent, Multi Resource swap: We have Agents 1, 2, 3, and Resources

of Resource Types

A, B, C representing any distinct Resources

.

All Input and Output Resources

in PTXs

are fully specified and persistent, with one party expressing a preference over the amounts they want to give away.

The CSP instance is generated from all the PTX

s which are chosen to be in the solving batch. Each Input/Output Resource

included in the PTX

will generate a variable

of the instance and each PTX

will generate a collection of constraints

.

For fixed Resources

, the domain

is equal to the Resource

. Mandatory balance checks imply domains

for variables

that are uninhabited on the other side of the Input/Output boundary.

If a Resource

is not fully specified, e.g. via an Ephemeral Resource

carrying a Predicate

, the domain

is defined by the Resource Predicate

(see later example).

The solver

then does a search for matches of Input and Output Resources

which satisfy constraint

. A PTX

gets included in a candidate TX

if the constraint

it represents is satisfied. Since Resources

from one PTX

can be assigned to Variables

from multiple other PTX

s, constraints of all PTX

s that share Resources

in their assignments must be satisfied simultaneously, otherwise none of the interdependent ones can be included in the TX

.

If the welfare_function

is linear or of low polynomial degree (e.g. quadratic), they can inform the solver about how to make better choices locally in the assignment, to raise the utility of an agent in the outcome. They can, in principle be free form, but if they do not lead to an efficiently computable gradient, increasing the utility for the agent will be less feasible.

// agent1 let have1 = Resource { res_type: "A", quantity: 4, owner: agent1, } let want1 = Resource { res_type: "B", quantity: 1, owner: agent1, }

// agent2 let have2 = Resource { res_type: "C", quantity: 2, owner: agent2, } let want2 = Resource { res_type: "A", quantity: 4, owner: agent2, }

// agent3 let have3 = Resource { res_type: "B", quantity: 4, owner: agent3, } let want3 = Resource { res_type: "C", quantity: 2, owner: agent3, } fn linear_preference3(tx: TX) -> scalar { / *maximize "B" with owner agent3 in the assignment, with utility linear in the amount of "B"* / } fn term3(tx: TX) -> TX { / *split up input resources, s.t. a remainder can be passed back to agent3* / } let ptx1 = PTX { input_resources: [have1], output_resources: [want1], }

let ptx2 = PTX { input_resources: [have2], output_resources: [want2], }

let ptx3 = PTX { input_resources: [have3], output_resources: [want3], welfare_function: linear_preference3(), } // Since ptx3 contains a linear preference, which is not in conflict with other constraints or preferences, it enables the solver to locally optimize the assignment to lead to a better outcome for agent3, according to their preference. // Should conflicts arise, they can be resolved by weighting whole PTXs, either by a random factor, by querying a trust/ranking database. Another option is to use rewards for the solver, by which a clause of the CSP instance induced by a specific PTX can be weighted.

// The above PTXs get submitted to the solver and it produces the following TX fulfilling their intents, after updating the PTXs as follows. let remainder_have3 = Resource { res_type: "B", quantity: 3, owner: agent3, } // balance check passes if in = [have3], out = [remainder_have3, want1] let ptx3' = Resource { input_resources: [have3], output_resources [remainder_have3, want3], term: term3, }

let tx = TX { ptxs: [ptx1, ptx2, ptx3'] }

Example: Multi Agent, Multi Resource Swap with non-fixed Resources and Linear Preferences

To further generalize the above, we introduce a preference over a partially determined resource bundle.

// agent1 let have1A = Resource { res_type: "A", quantity: 4, owner: agent1, } let have1B = Resource { res_type: "B", quantity: 2, owner: agent1, } fn linear_preference1(tx: TX) -> bool { / *prefer giving away 2.5A over 1B and sending the remainder back* / } let want1 = Resource { res_type: "E", quantity: 2, owner: agent1, } let ptx1 = PTX { input_resources: [have1A, have1B], output_resources: [want1], term: term1, welfare_function: linear_preference1() }

// agent2 let have2 = Resource { res_type: "C", quantity: 4, owner: agent2, } let want2 = Resource { res_type: "E", quantity: 2, owner: agent2, } let ptx2 = PTX { input_resources: [have2], output_resources: [want2], }

// agent3 let have3 = Resource { res_type: "E", quantity: 4, owner: agent3, } fn resource_preference3(tx: TX) -> bool { if tx.output_resources.contain(/ *(2A || 1B) && (1C || 1D) each with owner == agent2*/) { true } else { false } } let want_pref_eph_out3 = Resource { res_type: "preference for resource", quantity: 1, owner: agent3, predicate: resource_preference3() ephemeral: yes, } let term3 = Term { term: / *create want_pref_eph_in3 for balance check after matching* /, } let ptx3 = PTX { input_resources: [have3], output_resources: [want_pref_eph3], term: term3, }

// The solver searches according to the constraints encoded by the PTX , with additional constraints given by lin_pref_eph_in1 and res_pref_eph_in3 updates ptx1 as follows: let have1A' = Resource { res_type: "A", quantity: 2, owner: agent1, } let ptx1' = PTX { input_resources: [have1A], output_resources: [want1, have1B, have1A'], // have1B technically is a different resource after consumption and creation, but it is of the same type and amount term: term1, }

// ptx3 gets updated as follows, as indicated by the preference predicate and the term: let want3A = Resource { res_type: "A", quantity: 2, owner: agent3, } let want3C = Resource { res_type: "C", quantity: 1, owner: agent3, } let ptx3' = PTX { input_resources: [have3, want_pref_eph_in3], output_resources: [want3A, want3C, want_pref_eph_out3], term = term3, }

let tx = TX { ptxs: [ptx1', ptx2, ptx3'] }

Note on Tractability Research

The main source of conceptual complexity during solving will be in constraints encoded as Resource Predicates

of Ephemeral Resources

. A later post will elaborate on the classification of constraint languages

in respect to tractability of computation.