This is a copy/paste of a section I wrote for the upcoming Solver ART, filling out the ideas discussed at the Hacker House. This will hopefully make those ideas more clear to the parts of the organization it's relevant to. I'd like feedback (what's missing, what's unclear, what doesn't make sense, etc).

I'd particularly appreciate commentary from Game Theory oriented people on how to best view the problem of solving. I gave some rough gestures towards game theory aspects at the end, but I'm out of my element, there.

Abstract Resource Networks

In this section we will attempt to sketch one approach to formalizing the notion of a solver. We must start by setting the stage of solving in the form of an abstract description of a network of resources. We frame this in terms of a "transition relation" between network states. These are predicates over pairs of states which characterize what transitions are possible. We are not interested in a specific transition function; something which would give us the next state given the current. Instead, a transition relation may simply limit what transitions are allowed. We may, equivalently, view these as nondeterministic functions that can take a current state and return a set of next possible states. This coincides with earlier work on formalizing HPaxos in TLA+ and Isabelle. In that work, the actors ("learners" and "acceptors") don't have transition functions, just relations between subsequent states.

We will describe the network in different layers of complexity. In the first layer, we view the network as just a large set of resources which transform over time in accordance to user input, constrained by the logics of the resources. We formally define the state of the network as a pair of a set of user messages and a set of resources.

$$\text{NetworkState} := \mathcal{P}(\text{Msg}) \times \mathcal{P}(\text{Resource})$$

Where $\mathcal{P}$

denotes the powerset; the set of all subsets of its input. We will not assume further formal structure about Msg

. However, in practice, a Msg

can always be viewed as a serialized type (e.g. a bitstring from which information such as user identities might be parsed).

A Resource

will have additional data associated to it. For the simplest model, it will only have an associated Code

defining a relation between network states. We assume the following functions;

$$\text{ResourceLogic} : \text{Resource} \rightarrow \text{Code}$$

which extracts code for the "Resource Logic" of the resource; something that encodes a predicate between two states intended to identify valid state transitions, and

$$\text{Interp} : \text{Code} \rightarrow \mathcal{P}(\text{Msg}) \rightarrow \mathcal{P}(\text{Resource}) \times \mathcal{P}(\text{Resource}) \rightarrow \mathbb{B}$$

which interprets code as a relation between network states. Here $\mathbb{B}$

is the set of booleans, true

and false

. We will not make assumptions about the structure of Code

or its interpretation. In the resource machine report \textcolor{red}{CITE: Resource machine report}, Code

will be assembly code for Nockma, a virtual machine. However, this choice will not be important for the abstract description. Do note that a Code

must be interpretable as a predicate, rather than being an arbitrary function.

For the sake of succinctness, the following notation will be introduced to refer to the transition relation derived from a specific resource, r

, and set of messages, ms

;

$$\text{TR}^{ms}_{r}(s) := \text{Interp}(\text{ResourceLogic}(r), ms, s)$$

Our abstract description of the resource network seeks to describe the transition history as being made up of intuitively "atomic" transitions. There will be a transition for the arrival of messages and another, separate transition for the mutation of

resources. As such, a message cannot arrive simultaneous with a resource mutating. Note that this has nothing to do with time as transitions do not have times associated with them, just an ordering. As such, the relation defined by the resource logic can only be sensitive to messages already on the network when the transition happens, along with whatever modifications the transition makes to the resources.

Messages are also permanent. This is a consequence of the concurrent nature of the network. We cannot assume that a message will definitely be deleted as several resources may want to interact to the presence of a message long after that message is put on the network. We also do not want to allow the possibility of resources that only exist to “eat” messages from an arbitrary user.

With all this in mind, we can define the dynamics of the resource network as a transition relation. Firstly, we allow arbitrary messages to arrive at any time;

$$\text{MessageArrival} : \text{NetworkState} \times \text{NetworkState} \rightarrow \mathbb{B}$$

$$\text{MessageArrival}((ms_1, rs_1), (ms_2, rs_2)) := rs_1 = rs_2 \wedge \exists m : \text{Msg}.\ ms_2 = ms_1 \cup \{m\}$$

That messages can arrive at any time and have any content reflects the fact that these are inputs to the network whose provenance and motivation are alien to the network. As such, we make no assumptions by default that their history has structure.

Lastly, we may allow any transition which is allowed by modified resources on the network;

$$\text{ResourceTransition} : \text{NetworkState} \times \text{NetworkState} \rightarrow \mathbb{B}$$

$$\text{ResourceTransition}((ms_1, rs_1), (ms_2, rs_2)) := ms_1 = ms_2 \wedge \forall r \in (rs_1 \triangle rs_2).\ \text{TR}^{ms}_{r}(rs_1, rs_2)$$

where $\triangle$

denotes the symmetric difference. Intuitively, this states that any resource which is modified by a transition (that is, created or deleted) must approve of the transition doing so. Note that a resource which merely changes a parameter, for example being traded between users, will be two resources, one owned by the original user storing the id of that user which is deleted, and another owned by the new user which is created. Resources which are unmodified do not need to approve of a transition. This means one cannot create a resource that halts the entire network by disapproving of all transitions. This also means it’s always possible for a network transition to simply do nothing as such a transition does not need to seek approval from any resources.

Intents on Abstract Resource Networks

It’s up to resource designers to create adequate resource logics for whatever application they seek. This report is about solvers and intents, so it is prudent to consider how that manifests in this framework. Intents are special cases of resources. There are several ways these may be set up; we will describe one.

Consider the case of a simple swap. We have two resources on the network, call them A

owned by user 1 and B

owned by user 2, and the owners of their resources want to trade but they don’t know about each other. To seek the trade, both users create intents whose resource logic requires a matching intent along with the availability of the resource involved with the matching intent. Simultaneously, the intents annihilate with each other along with A

and B

, creating two new resources, B

owned by user 1 and A

owned by user 2. This is illustrated in the following figure.

[

Screenshot from 2024-09-26 11-30-52

984×884 59.2 KB

](https://europe1.discourse-cdn.com/flex013/uploads/anoma1/original/1X/ecbb493d0f41a1dea6df75ddf4b68b2c4e7e675c.png)

The intents resource logic should govern the creation and deletion of intents. An intent may allow its own creation so long as

a message exists on the network requesting it.

Without any further conditions, an intent may be created an unbounded number of times since the message will continue to exist. There may be other conditions used to ensure an intent isn't created multiple times. This is a design choice with many solutions. Intents may require that the resources they seek to trade still exist. They may have ids and a condition that an intent cannot be created with an id still in use. Intents may have an expiration condition that references a clock-time resource used to keep track of time, approximately. They may also check if a "delete intent" message is present which cancels out the action of the original intent creation message.

Once the intents are present, they will wait for messages that give a match. Such messages may come from anywhere, but they will come from a "solver", which is just an ordinary user from the perspective of the network. When an appropriate solution message is present, the transition functions which delete the intents and swap the resources can be made.

It's up to the resource designers to construct their logics in such a way that only what they want to happen can happen. Additional conditions, such as the presence of cryptographic keys that prove ownership, will likely be present to prevent unapproved swaps.

Another variant of this setup won't have the intents doing the swap, but, rather, the swap will happen only in the presence of approval messages. What a solver will do, then, is create a "solution" resource which deletes the intents but doesn't swap the resource. The owners of the original A

and B

resources will then need to supply permission messages for the solution resource to delete itself and actually perform the swap.

This general framework is very flexible, and does not force a single solution to any given problem.

Controllers on Abstract Resource Networks

We may specialize this description with additional assumptions to incorporate more structure. In many of the networks we care about, resources are manipulated by "controllers". All the resources in the network are in the view of a single controller, and resources on different controllers can't interact. We can start describing this by assuming each resource has an associated controller:

\text{ResourceController} : \text{Resource} \rightarrow \text{ControllerId}

where \text{ControllerId}

is some countable type used to uniquely identify different controllers. Given a set of resources, rs

, define the set of resources inside and outside some controller associated with the resource, r

, using the following notations;

\star|_{r \in} := { x\ |\ \text{ResourceController}(x) = \text{ResourceController}(r) }

rs|_{r \in} := { x \in rs\ |\ \text{ResourceController}(x) = \text{ResourceController}(r) }

\star|_{r \notin} := { x\ |\ \text{ResourceController}(x) \neq \text{ResourceController}(r) }

rs|_{r \notin} := { x \in rs\ |\ \text{ResourceController}(x) \neq \text{ResourceController}(r) }

We can assert that only resources on the same controller can interact by assuming these statements;

Formula 1:

\forall ms, r, rs_1, rs_2. \ \text{TR}^{ms}*{r}(rs_1, rs_2) \rightarrow rs_1 [*r \notin} = rs_2 |_{r \notin}

Formula 2:

\forall ms, r. \ \forall o_1, o_2 \in \mathcal{P}(\star|*{r \notin}). \ \forall i_1, i_2 \in \mathcal{P}(\star|{r \in}). \ \text{TR}^{ms}*{r}(i_1 \cup o_1, i_2 \cup o_1) = \text{TR}^{ms}*{r}(i_1 \cup o_2, i_2 \cup o_2)

Together these state that the interpretation of the resource logic of any given resource will truly be a relation over the resources on the same controller, and the relation over all resources on the network will be that relation extended to be the identity relation on all other resources. Formula 1 states that a resource transition can only modify

resources on the same controller, but it does not disallow transitions which are sensitive to the presence of resources on other controllers. For example, under this assumption a resource might delete itself so long as a different resource is present on a separate controller. To prevent that, formula 2 states that changes in the resources "outside" the controller don't affect

whether a resource logic is satisfied.

We may observe that any transition function of a resource specializes to a transition function over only the resource on its controller;

\text{TRC}^{ms}*{r} : \mathcal{P}(\star|*{r \in}) \times \mathcal{P}(\star|_{r \in}) \rightarrow \mathbb{B}

\text{TRC}^{ms}*{r}(s) := \text{TR}^{ms}*{r}(s)

Here we are coercing the s \in \mathcal{P}(\star|_{r \in})^2

into a \mathcal{P}(\text{Resource})^2

by interpreting the outside resources as being empty. We may further prove that

\forall ms, r. \ \forall rs_1, rs_2 \in \mathcal{P}(\text{Resource}). \ \text{TRC}^{ms}*{r}(rs_1 |*{r \in}, rs_2 |*{r \in}) = \text{TR}^{ms}*{r}(rs_1, rs_2)

as a corollary of formulas 1 and 2 by considering the case where o_1 = \varnothing

and o_2 = rs_1 |_{r \notin}

.

Solving

Solving occurs as a consequence of attempting to optimize the network. We assume a user has a preference for different network states expressed through a utility function;

U : \mathcal{P}(\text{Resource}) \rightarrow \mathbb{R}

We may first attempt to formulate the problem of a salver as the goal of sending messages to network such that the value of U

is maximized in expectation. The model of our network has no probabilities assigned to the transitions, meaning it's not possible to calculate expectation without additional assumptions about the likelihood of different transitions.

The network will constantly be gaining new messages from other users. The concurrent nature of the network and the unknown source of messages makes it difficult to establish a prior distribution modeling the system. As such, abandoning a prior assumption may be prudent. Without those additional assumptions, there's no way to define a notion of expected utility.

We may instead attempt to view the system as an online learning problem where the solver may observe the network as much as they want, and the solvers actions are the messages they may send. We may assume a restricted set of messages related to solving some notion of intents if we want to narrow down the job of a solver as something distinct from a regular user.

We may try to frame the problem the solver is solving as a regret minimization problem. At some point in time, t_0

, it starts observing the network. At each time step (which is not necessarily the steps the network itself takes) the solver makes an observation of the network, and makes an action, which is either nothing or the sending of a message. This produces a history of observations, t_i

, where i \in {1..T}

, where T

is the current step.

Under a standard regret minimization formulation, we'd seek to compare the solver's cumulative utility to the utility obtained by the best fixed action. However, our setting has an environment which reacts to the solver's actions. The solver taking a different action in the past may change the utility it would observe after the same action in the present. As such, it would not be feasible to assess the "best fixed action in hindsight". We may attempt to formulate this in a regret-free formulation of online learning, such as that presented in \textcolor{red}{CITE: Mechanisms for a No-Regret Agent}. In such a setting, we imagine our setting as a Stackelberg game, where the network is viewed as "nature", and the source of non-solver messages is viewed as the principle, setting policy which the solver must respond to. The solver's goal is then to formulate a policy which minimizes the difference between the achieved utility and the best possible utility under any

policy.

We may also formulate this in simpler terms, merely viewing the solver as a single player responding to a, possibly adversarial, environment. We may formulate its goal as a minimax strategy, where its goal is to find a policy that maximizes

the minimum cumulative utility under worst-case assumptions about environmental response.