Course: CS4710
Chanpech Hoeng

Abstraction
a. Identify abstractions you want to have in your Alloy model and declare them as signatures.

First approach signatures:
1. ID
2. Entity - Abstract
3. Sender
4. Receiver
5. FinancialInstitution
6. Transaction
7. Currency

Last approach signatures
1. Institution
2. Funds
3. ID
4. Account
5. FinancialInstitution

b. Determine which signatures should be declared as abstract sig, and explain why.

First approach:

```
abstract sig Entity {
  id: one ID,
  balance: some Currency
}
```

Entity is the only one that is declared as an abstract signature. To capture the relationship of the account the Sender, Receiver, and Financial Institution must all have ID and Currency. So to reduce the complexity and increase the efficiency of our programs it makes more sense just to generalize these attributes in the Entity signature and make it inheritable.

Relations
Specify the relations between the signatures you define in the previous step.

The abstraction between the Entity and its extended signatures has the following relationship defined by the following predicates.
pred TransactionProcessing[t, ut: Transaction] {...}

pred TransactionProcessingHelper[t, ut: Transaction, p1: Person1, p2: Person2] { … }
pred SystemTransition[t, ut: Transaction] {... }
Person1 and Person2 whose inherited attributes of Entity will be able to initiate transactions defined by the TransactionProcessing predicate. While in FinancialInstitution while being invoked as a middleman during this transaction by the TransactionProcessingHelper.

Multiplicities
Specify the multiplicity of each signature participating in a relation. (10 points)

Ideally under the hood, the process of transaction that happened between Person-x to Person-y should be one to many as allowed by the balance in their account. On the other hand, the financial institution should have many to many relationships during its transaction processing signature. However, during this initial part, we will narrow down the multiplicity to make our program more achievable. This means that the Person and Financial Institution will have a one-to-one relationship throughout the transaction.
For example, our program will try to simulate, Person1 will try to initiate a transaction with Person 2. However, behind the scenes, the financial institution will act as a link that moves the transaction from Person1 to Person2. Thus 1-to-1 throughout.

Incremental modeling
a. Please include the final three incomplete versions of your Alloy model that you incrementally improved. Explain the differences between each version with respect to its previous version and why you improved it. (15 points)

- *Version 1 signatures and basic predicate:*

```
module OPS

//Utilize dynamic modeling
//Library for ordering located in the util folder
open util/ordering[ID] as ord //Similar to a doubly linked-list

sig ID{}
abstract sig Entity { id: one ID }

one sig Sender, Receiver, FinancialInstitution extends Entity {}

sig Currency{
	amount: Int
}

sig Transaction {
	receiver: set Entity,
```

```
        recipient: set Entity
}

pred move[t, ut:Transaction ]{
        FinancialInstitution in t.receiver implies FinancialInstitution in ut.recipient && FinancialInstitution
not in ut.receiver
                else//He has not yet been in the south
                FinancialInstitution in ut.receiver && FinancialInstitution not in ut.recipient

        some e:Entity - FinancialInstitution  |
                FinancialInstitution in t.receiver implies (FinancialInstitution + e) in ut.receipient &&
(FinancialInstitution + e) not in ut.receiver
                        else (FinancialInstitution + e) in ut.receiver && (FinancialInstitution + e) not in
ut.receipient
}
```

- *Version 2 Implementation of facts :*

```
module OPS

// Utilize dynamic modeling
// Library for ordering located in the util folder
open util/ordering[Transaction] as ord // Similar to a doubly linked-list

sig ID {}

abstract sig Entity {
  id: one ID,
  balance: lone Currency //Can have a currency or nothing at all
}

// Generalize
one sig Person1, Person2, FinancialInstitution extends Entity {}

sig Currency {
  amount: Int
}

sig Transaction {
  sender: set Entity,
  recipient: set Entity
}

// Facts and Predicates

fact UniqueIDs {
  all disj e1, e2: Entity | e1.id != e2.id and e1.balance != e2.balance
}
```

```
fact OnlySenderReceiverCanTransact {
  all t: Transaction | {
    some p1: Person1, p2: Person2 | p1 in t.sender && p2 in t.recipient
  }
}

//fact NoSelfTransactions {
//  all t: Transaction | no p: Person1 | p in t.sender && p in t.recipient
//  all t: Transaction | no fi: FinancialInstitution | fi in t.sender && fi in t.recipient
//}

fact CurrencyTransfer {
  all t: Transaction |
    let sender = t.sender,
        recipient = t.recipient |
      sender.balance & recipient.balance = sender.balance
}

fact InitialBalances {
  all e: Entity - FinancialInstitution | lone e.balance
}

fact NoTransactionWithoutCurrency {
  all t: Transaction | some sender: t.sender | lone sender.balance
}

pred TransactionProcessing[t, ut: Transaction] {
  some p1: Person1, p2: Person2 | TransactionProcessingHelper[t, ut, p1, p2]
}

pred TransactionProcessingHelper[t, ut: Transaction, p1: Person1, p2: Person2] {
  (p1 in t.sender && p2 in t.recipient) implies {
    ut.recipient = t.sender && ut.sender = t.recipient + FinancialInstitution
  }
}

pred SystemTransition[t, ut: Transaction] {
  TransactionProcessing[t, ut] or TransactionProcessing[ut, t]
}

run SystemTransition for 5 but exactly 3 Entity, 2 Currency, 3 Transaction
```

- *Version 3: The model didn't achieve what we wanted. So we decided on a different approach.*

```
module OnlinePayment

sig Institution{}
sig Funds{}
```

```
sig ID{}

sig Account{
	id: one ID,
	balance: set Funds,
	send: lone Account,
	receive: lone Account,
	registered: one FinancialInstitution,
	spentFunds: set Funds
}

//sig User extends Account{}
one sig FinancialInstitution extends Institution{
	processSenderID: one ID,
	processReceiverID: one ID
}

//All account will have a Unique ID and unique funds
fact Unique {
  all disj u1, u2: Account| u1.balance != u2.balance and u1.id != u2.id
}

//An account can't send and receive to itself
fact noReflexiveSendReceive{
 all a: Account| a.send != a and a.receive != a
}

//Since an account won't be sending and receiving to itself, the finacial institution shouldn't process
// a case where processSenderID are the same as processReceiverID.
fact noReflexiveSendReceive{
 all f: FinancialInstitution, a:Account | f.processSenderID = a.id implies f.processReceiverID != a.id
}

//An account can only spent the funds if and only if that fund have a send attribute
//fact spentOnlyWithSend {
//    all a: Account, f: Funds | f in a.spentFunds implies a.send
//}


//If there are accounts that share the same funds then one of the account must have initate send.
pred sameFundsImpliesSendReceive[a1, a2: Account] {
   a1.balance = a2.balance implies a1.send = a2 and a2.receive = a1 ||
	a2.send = a1 and a1.receive = a2
}

//An account can send but it does not have an incoming receive from other account
pred sendWithoutReceive[a1, a2: Account] {
   a1 != a2 implies (a1.send = a2 implies a2.receive != a1)
}
```

run shows{}
//
//run exactly 2 Account, 2 Funds, 1 FinaicialInstitution

The first version takes a lot of inspiration from the farmer riddle problem. We follow a very similar approach to the abstraction and signatures. For the financial institution, we pictured it as a middleman that works to handle the transaction similar to how the river takes the farmer from one place to another.

In the second version, the following predicates and facts are added to enforce a constraint that allows only the Sender (Person1) and the Receiver (Person2) to be the only one to initiate the transaction.

fact OnlySenderReceiverCanTransact { … }

Then I also introduced the following predicates that dictate how the transaction can be processed between the two entities with the facilitating from the financial institution. The financial institution plays a role as the middleman by acting as a placeholder for the recipient during the updated transaction.I also introduce new facts that would eliminate reflexive transactions. In summary, this fact states that for all transactions that happen between each entity, there can't be a case where they are their own sender or their own recipient (*fact NoSelfTransactions {...}*).

In addition to NoTransactionWithNoCurrency and CurrencyTransafer. However, currently, the transaction is working fine but the state of the currency is not being transferred upon transaction.

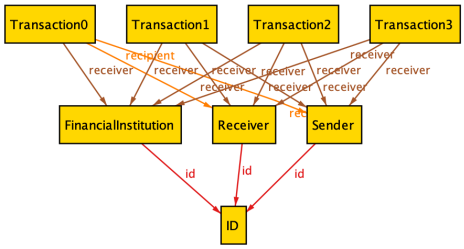
pred TransactionProcessingHelper[]{...}

pred TransactionProcessing[]{...}

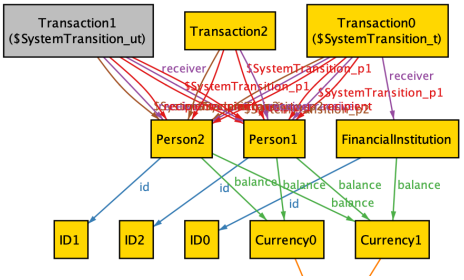
In the third version, I decided to take all of my previous knowledge and start from a different approach. In the previous version, I overcomplicated a lot of the facts and predicates simply due to my ignorance of the alloy. But after seeing my classmate's approach I decided it would be best to redo my approach. Thus For this version, I have defined the following set of predicates: If there are accounts that share the same funds then one of the accounts must have initiated send would be demonstrated through the pred sameFundsImpliesSendReceive[a1, a2: Account] {... }. An account can send but it does not have an incoming receive from other accounts would be demonstrated through pred sendWithoutReceive[a1, a2: Account] {... }

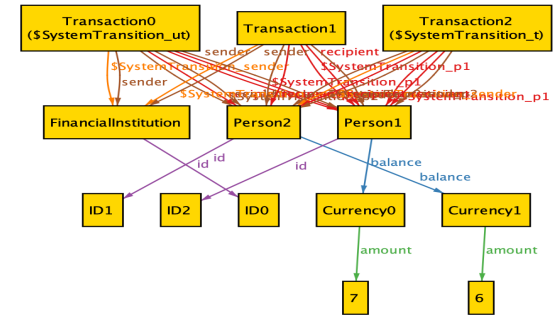
b. Include some snapshots of the instances you visualize for each version. (5 points)

- Version 1:

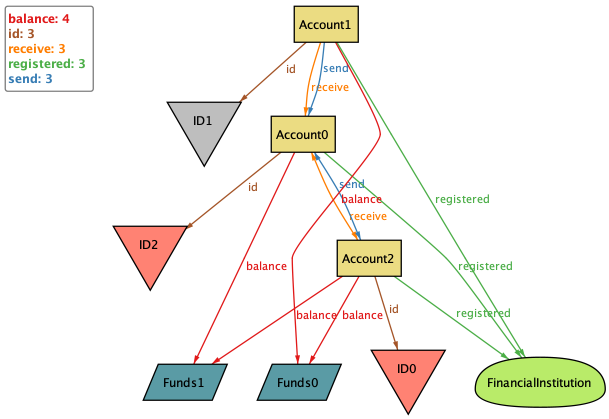


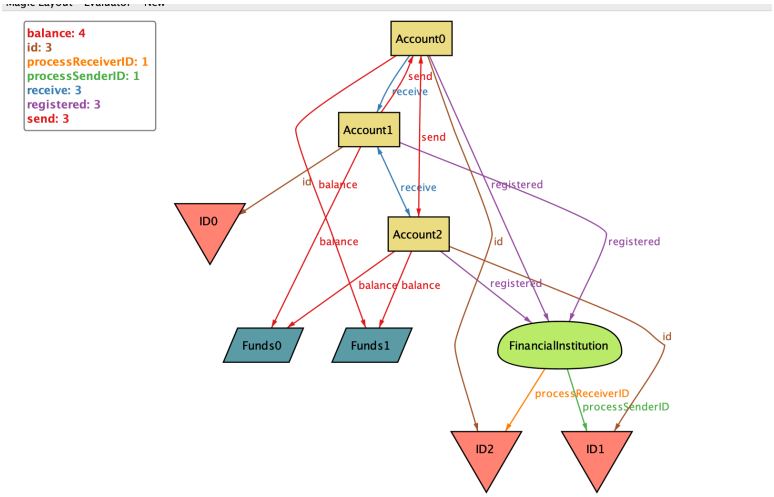
- Version 2:





- Version 3:





this/Account	id	balance	send	receive	registered
Account⁰	ID²	Funds¹	Account²	Account²	FinancialInstitution⁰
Account¹	ID¹	Funds⁰	Account⁰	Account⁰	FinancialInstitution⁰
Account²	ID⁰	Funds⁰	Account⁰	Account⁰	FinancialInstitution⁰
		Funds¹			

Facts

I started to introduce the following facts in version 2:

```
fact UniqueIDs {...}
fact OnlySenderReceiverCanTransact {...}
fact NoSelfTransactions {...} //but it seems to find no instances due to this thus I have commented it out.
fact CurrencyTransfer {...}
fact InitialBalances {...}
fact NoTransactionWithoutCurrency{...}
```

In my final version:

```
//All accounts will have a Unique ID and unique funds
```

```
fact Unique {
  all disj u1, u2: Account| u1.balance != u2.balance and u1.id != u2.id
}

//An account can't send and receive to itself
fact noReflexiveSendReceive{
 all a: Account| a.send != a and a.receive != a
}

//Since an account won't be sending and receiving to itself, the financial institution shouldn't process
// a case where processSenderID is the same as processReceiverID.
fact noReflexiveSendReceive{
 all f: FinancialInstitution, a:Account | f.processSenderID = a.id implies f.processReceiverID != a.id
}

//An account can only spend the funds if and only if that fund has a send attribute
fact sendImpliedSpentFunds {
   all a: Account, f: Funds | f in a.spentFunds implies a.send
}
```