FORMAL MODELING OF BANKING POLICIES USING ALLOY ANALYZER

TECHNICAL REPORT

Jeremy Johnson
Syracuse Universityt
jjohns40@syr.edu

Izzat Alsmadi
Texas A&M, San Antonio
ialsmadi@tamusa.edu

October 10, 2021

ABSTRACT

Any business domain in the world attempts to set up their business in a logical manner and attempts to ensure people can not take advantage of it. The Alloy language is going to be used to model the banking domain and prove that you can make policies that make sense and do not contradict each other. Policies in this paper have three primary levels, the “bank", the “User" and the “Accounts". Banks have Users and Users have Accounts, exactly how you would expect it to be in the real world. The emphasis for the security of the banking policy is derived from how each Alloy signature connects to their counterparts. Facts are used heavily to constrain the model to limit interactions between the nodes, checks and assertions are used to ensure there are no security vulnerabilities in the model. To expand the valid operations within the bank we added Time and State libraries, this enables us to view the model from one point in time to the next and observe how operations affect the model.

1 Introduction

Modeling software development problems has been taking place ever since computers were invented. With recent advancements people have formalized the approach to creating and verifying these models with the Alloy language. Alloy uses a SAT solver to quickly examine a subset of states that the model can take and creates valid states visualizes it for the user in a GUI. This new technology enables software developers to quickly test a model before using precious time programming the system only to find out there is an integral flaw in how the system is set up. This project uses Alloy to create a model based on the banking industry and uses Alloy’s features to prove that the model is secure. Our model covers static and dynamic, using facts, predicates, assertions, time and states to constrain and prove the security of this banking model.

2 Banking Policies Used

This project evolved through out the class and was continually refined through each project. Initially the model was focused on the users and their relations, having parents, children, marriage, etc. This was quickly fixed through feedback to focus on the bank; by abstracting away these details and leaving just "users" it left more room to focus on components of a bank such as accounts, regular and privileged users at the bank. The second major iteration of the project fixed these issues and was well constrained with facts. This was a step in the right direction, however the model lacked operations that users can normally perform within a bank. To enable this we expanded the model to include a bank card, a PIN, and an account value. These entities and values were exactly what was needed for the users to deposit money and interact with the bank. That brings us to the current and final dynamic iteration of the model, it is well constrained and contains attributes that enable a plethora of operations to take place within the model. This transformation can be seen in the figure below, the left image is the first draft and the right image is from the final static model. The signatures were improved and the rules constraining which interacts with which creates a far cleaner and simplified model.

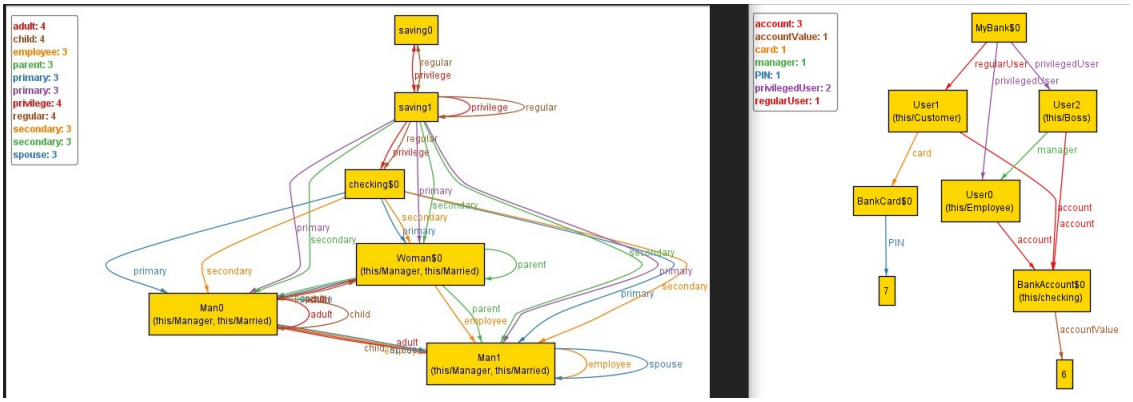

Figure 1: Left Image: Project 2, Right Image: Static Final Project

The bank has two types of users, “regular" and “privileged". The type of user will dictate the type of connection the user gets from the bank. For example, if the user is a “Customer" then they will always get a “regular" connection to the bank. Conversely if the user is an Employee or Boss they will get the “privileged" connection to the bank. This ensures that the proper users are always connected in a secure and regulated way to the bank. There can be three types of users, namely customers, employees, and bosses. Each employee has a boss that they report to through the “manager" connection, this ensures a valid chain of command within the employee hierarchy at the bank. All users at the bank can open a checking or savings account. Between any two customer accounts you can have the shared account trait, implying that the users have access to both accounts. Users at the bank have the "card" extension that leads to the signature "BankCard". This Bank Card belongs to all customers in the bank, each bank card also has a PIN which is type Int. Due to Alloy language constraints we can not require this PIN to be 4 digits, however we do require it to be greater than zero. Each account also has an account value, including this as a trait of the BankAccount allows me to perform actions such as deposit money. To help illustrate all major signatures in the model we have attached the image below, the image is structured logically with the bank being at the top, followed by the User signature and lastly the Account signature.

```
one sig MyBank {
    regularUser: set User -> Time,
    privilegedUser: set User -> Time,
}

sig User {
    account: set Account -> Time
}

sig Boss in User {
    manager: set Employee -> Time,
}
sig Employee in User {}
some sig Customer in User {
    card: set BankCard -> Time,
}
sig BankCard {
    PIN: Int
}

abstract sig Account {}

sig BankAccount extends Account {
    accountValue: Int
}
sig checking, saving in Account {}
sig SharedAccount in Account {
    shared: one Account
}
```

Figure 2: All major signatures in Alloy Model

To ensure that all aspects of the model hold true there are many facts users through out the Alloy program. Facts within the set of the users includes that no user can share an account with themselves and no user can have both traits of Boss and Employee at the same time. Sharing accounts is restricted exclusively to customers and customers are always going to have a regular account with the bank. Within the accounts we enforce that all saving and checking accounts are connected to users at all times. Allowing non-connected accounts would be a security risk for the user who deleted it. we also require all Bosses and Employees at the bank to have privileged accounts, this allows them to have the ability to do their jobs and prevents nefarious customers from obtaining permissions they are not supposed to have.

Assertions are used to prove that an instance of a model is extremely unlikely to happen. When designing our assertions we wanted to prove that the core aspects of the security within our model were logically sound. We successfully tested that no employees or bosses at the bank can share an account, no users can be bosses and employees at the same time, and that all customers are regular users at the bank. To prove this we used the “check" Alloy command, because Alloy did not finding counter examples we are confident that the scenarios are not possible. The Alloy code in the image below is used to prove one of the most important security requirements in our model, stating that no user can be a regular user and privileged user at the same time. The code iterates over all users and banks, stating if the user is a regular user that this implies that the user is not a privileged user. The Alloy code attempts to create a counter example where this situation exists and fails to do so.

```
assert no_users_with_dual_statuses {
    //read as: For all banks, and for all users, each user has either regular user OR privileged user, but
    //             not both. Having both would violate a basic security policy.
    all b: MyBank | all u: User, t:Time | u in b.regularUser.t implies u not in b.privilegedUser.t
}
check no_users_with_dual_statuses for 5
```

Figure 3: No users with dual statuses

clicking "Execute All" in Alloy results in running all of the check commands in our model. The image below is spliced together to show all check commands finding "No counter examples". The image above is number 31 in the output below.

```
#24: No counterexample found. all_customers_regular_users_all_time may be valid.
#25: No counterexample found. zero_or_more_customers may be valid.
#30: No counterexample found. accounts_no_self_share may be valid.
#31: No counterexample found. no_users_with_dual_statuses may be valid.
#32: No counterexample found. all_customers_are_regular_users may be valid.
```

Figure 4: No Counter Examples Found

Predicates are useful for asserting a condition that returns true or false. For example, when provided with a user and a checking account, the predicate states that the checking account is in the users account. This is also used in our model to state that employees and bosses are not in a shared account, and that for a given boss employee pair the boss is the manager of the employee. The image below shows an example of a predicate that initializes two users using an "initialization predicate". Once initialized it states that for the two users u and u’, if u is an Employee and u’ is a Boss then u has u’ as a manager. It is important to note that this is not saying this is true for all users, just these two distinct users that we are provided with in this scenario. For the sake of completeness we have included the Alloy code for initializing a user in the image as well.

```
pred init_user (u:User, t:Time) {
    //Employes >= Boss ensures there are no extra bosses at the bank. Also PINs need to be > 0.
    #Employee >= #Boss
    u.card.t.PIN > 0
}
pred boss_are_managers_for_employees[u:User, u':User, t:Time] {
    //read as: for the provided boss and employee, the boss is the employees manager.
    init_user[u,t]
    init_user[u',t]
    //read as: for the provided boss and employee, the boss is the employees manager.
    u in Employee and u' in Boss implies u in u'.manager.t
}
run boss_are_managers_for_employees {} for 5
```

Figure 5: Predicate Example

Functions can be used to return sets of items within the model. We used functions to identify a group of accounts between two users, the set of employees between two bosses, and the set of all users in the bank that have a regular account. These functions are extremely practical to have in the real world, being able to query how many regular users are in a bank or get the set of all employees between two bosses is information that can be used to drive decisions in the workplace.

To expand what our model is capable of we have included the "Time" utility. This lets me examine two different moments in time and impose rules from one time to the next. For example we can add a new user by stating that the user is not in the bank at time t, however in the next time t' we state that the user is in the bank. This is a simplified version of how you execute pre and post conditions in the Alloy language model. We used this to create models for adding customers over time, show users opening accounts over time, show users depositing money, and creating a new PIN. We also added the operations for withdrawing money, hiring employees and lastly firing employees. Each of these required two predicates, the first establishes the number of users, bank accounts, time, etc that we will be working with. The second is where we state the pre and post conditions required to make the model work as required. In addition to this we have established "CRUDs" for each major signature that we have in our model. A "CRUD" is an acronym for Create, Read, Update, and Delete. Each of these are actions that can be performed on the signature in the model. For example, for our "BankCard" the Create enables the user to make a new card, Read returns the current user PIN, Update lets the user change the PIN, etc. Each action relates to the signature in a logical way that you would be able to perform in the real world. The image below is a graphical representation of how Alloy displays our model with Time. This specific image is from running the unique action to "delete account" in the bank.

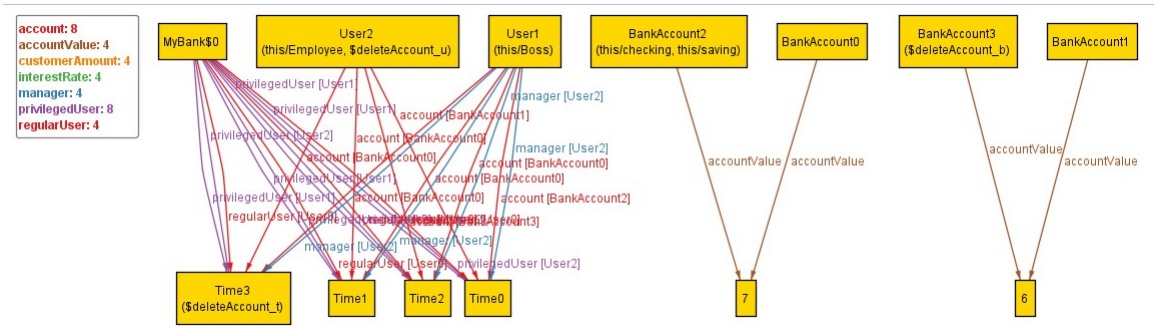

Figure 6: Operation - Delete Account With Time

The final major portion of this project is the inclusion of "State" and ordering within it. When used with the "ordering" Alloy library you are able to model arbitrary states from the beginning to end from one state to the next using the "next[s]" syntax, where "s" is an instance of the state. To emulate different states we created a state with the current

interest rate and the number of customers the bank currently has. These are integers that lend themselves to easily being increased, decreased, or cleared outright in operations. The image below shows the declaration of the Time and State signatures that are used throughout the dynamic code.

```
open util/ordering [Time] as T
open util/ordering[State] as so
sig Time {}
sig State {
    customerAmount: Int,
    interestRate:Int
}
```

Figure 7: Time and State Declarations

The State portion of our model is showcased by utilizing fact traces, regular predicates and "predicate trans" in the alloy language to model the operations. These operations simulate real world scenarios, and utilizing them enable me to iterate over all states performing these iterations at each different state. For example, "predicate trans" allows me to call a multitude of functions at each state as Alloy traverses from the first state to the second to last state. The image below shows the Alloy code for iterating through two States, s and s' and also shows the output model that Alloy provides.

```
//This is a predicate trans for the State interestRate
pred interest_trans[] {
    //for some State s and s', we add an interest point, decrease by one point, or clear interest completely.
    (some s,s':State | remove_interest[s,s'])
    or (some s,s':State | decrease_one_interest_point[s,s'])
    or (some s,s':State | increment_interest[s,s'])
}
run interest_trans for 4
```

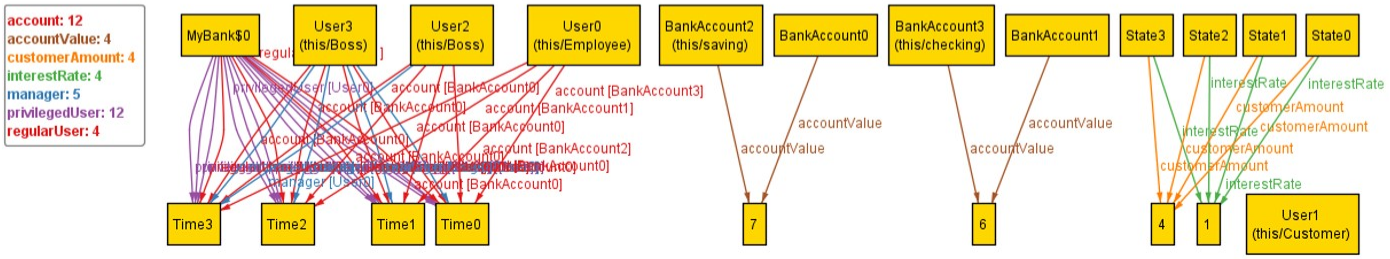

Figure 8: Pred Trans Code and Model

3 Conclusion

Banks move immense amounts of money every day and their security is vital to a successful country. Utilizing the Alloy programming language we have shown how to generate a simple and secure instance of a bank. By carefully crafting rules that all users and employees must abide by we are able to examine different scenarios to ensure a secure banking system. We used tools provided by Alloy including facts, assertions, checks, predicates, functions, ordering, Time and States. We used these tools to show the hierarchical relationship between the bank, the users, the accounts and the special relationships between them all. Using Alloy's time utility we were able to add actions to the bank allowing the users to perform actions you would normally be able to in the real world from one moment to the next. This enables us to model the banking system in a dynamic fashion very similar to how the real world operates and prove the model is secure in the process.

4 Related Work

4.1 Source 1 - A UML Class Diagram Analyzer

This article runs us through three iterations of the banking model. The first iteration of the model implemented in this paper has a "side-ways" relationship in stark contrast to our top-down model. By "side-ways" we mean the central unit will have multiple different types of signatures leaving it, namely the bank is has "Accounts" and "Customers". In our approach we had banks have users and users have accounts, from top to down. This helps provide a direct link from the user to the account within the bank, leaving no question which accounts belong to which users. In figure 1 below taken directly from source 1, you can clearly see the account as disjoint from the customer. The paper and we both agree that this approach does not model how banks operate in the real world and proceeds to run through two more iterations to improve it. The second iteration of the model adds additional details to the model, namely saving accounts, customer ID and a bank card. Our first thought when seeing this was how the ID could have been used as a direct and unique link to the customers accounts. This way the ID would operate similarly to a key in a python dictionary, mapping to a value. For the final third iteration the model officially connects the accounts to customers using the "owned_accs" connection. With this change we consider the model to be correct in relation to the real world. In our own spin on this model we would have made the banks to have customers, and then customers have the ID, bank card, and accounts. After reading this paper we decided to do exactly that, adding to our model an ID and BankCard signature coming from the user. It is a welcome addition to our model and adds a layer of realism.

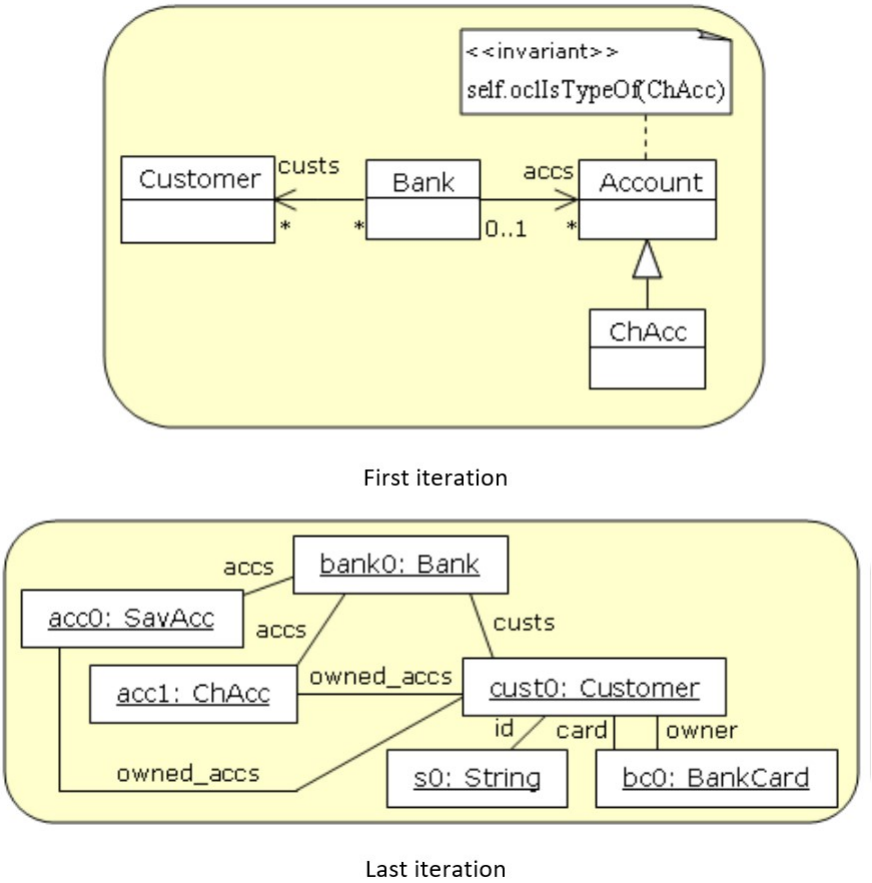

Figure 9: Image of First and Last model from Source 1

4.2 Source 2 - Formalization of Web Security Patterns

This article focuses primarily on how the customers interact with the banking system. This has a class called “operations" and it allows the users to have traits like fundraisers, bill payments, logging in, authentication, etc. This is a very detailed and focused approach to the banking policies and actions that the user can perform. Our model is much more simplistic in comparison, allowing for employees, accounts, and roles within the bank. Our model has no notion of what sort of actions each member of the bank can perform based on their roles within the bank, instead focusing on security aspects and how they interact with each other. Further in the article it discusses properties of the banks, threats to the bank, and patterns that it looks for. This is a very in-depth approach to examining potential security vulnerabilities in a banking system and checking for vulnerabilities in the model, however we were unable to find any mention of the insider threat with people who work at the bank. The latter of this paper focuses on secure frameworks for the customer to interact with the bank, however it doesn’t mention employees and the importance of keeping their accounts separate from customers. Our model specifically separates employees accounts from employees accounts to ensure customers can not access privileged information.

4.3 Source 3 - Software Analysis and Verification — Miniproject

This article discusses “JAHOB” proof obligations and feeding them into Alloy models for verification. The exact usage and application of JAHOB is beyond the scope of our paper today, but it speaks to the versatility of Alloy’s SAT-based bug finder. This paper takes the approach of focusing on the bank account balances within the accounts. The paper uses many unique facts for the checking account that help the user deal with amounts inside of the account. For example, there is an "amount" variable that has the type “Int". By defining the checking account to have a type “Int" it allows the model to examine states of the checking account with different numerical values within it. This is a layer deeper than our model previously considered, our model allows for accounts like checking and saving but does not consider states the account can take with various account values. This idea could be expanded to calculating the account value, being allowed to deposit a check, overdraft fees, etc. The article also discusses assertions and checks against the area of banking it is focused on. The code shows a proof written in both JAHOB and Alloy, which then shows Alloy finding no counter examples. This line of thinking is identical to how we used Alloy to prove that security features within our model have no counter examples.

4.4 Source 4 - A Static Semantics for Alloy and its Impact in Refactorings

This article does a great job discussing Alloy, what it can do, and then provides examples with a banking policy. The model the article creates is different from mine in how they chose to organize the bank, accounts, and customers. The model has the bank with both accounts and customers going from it. This is almost identical to the third stage of model shown in source 1, only missing the bank card and ID. When creating our model, we went with having the bank have users, and users having accounts. We prefer our approach to this problem because the relationship between users and the accounts is clear, the customers have accounts. With the paper’s setup it shows the “Owner” pointing to the customer, almost as if the account owns the customer. This can be clearly seen in Figure 2 below taken from the source. The paper continues to focus on accounts and the actions that the users can perform with them. It shows actions such as having a bank account balance, withdrawing money and how to handle the account balance in these situations. Our model stops short of having account balances, choosing to abstract away the bank account balances and just show the checking and saving accounts.

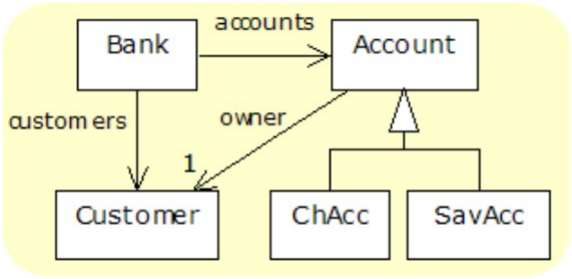

Figure 10: Image of the model in Source 4

4.5 Source 5 - Enabling Verification and Conformance Testing for Access Control Model

This article discusses how important choosing the correct model is by showing a banking system with conflicting job roles and how to use facts to solve the conflict. This approach to use facts to constrain the model is one approach, however in our experience we have found re-organizing the entire model is often more appropriate. Initially we claimed that accounts had users, and no matter which facts we attempted to use we could not identify the correct constraints to solve the problem. In the end changing the hierarchy was the correct approach for me, ensuring that users have accounts and not the other way around completely solved our issues. The model that the article explores the banking system with customer service representatives and loan officers. It explores the initialization of the two roles and uses a fact to state that having both of these roles is in the conflicting role set in the model. These roles are interesting categories of employees at a bank, we chose to abstract away specific duties in our model and label all users generically as Employee or Boss.

4.6 Sources

1. Massoni, T., Gheyi, R. and Borba, P., 2004. A UML Class Diagram Analyzer. [online] Citeseerx.ist.psu.edu. Available at: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.5763&rep=rep1&type=pdf> [Accessed 26 February 2021].
2. DWIVEDI, A. and RATH, S., 2021. Formalization of Web Security Patterns. [online] Available at: <https://www.researchgate.net/publication/281279993_Formalization_of_Web_Security_Patterns> [Accessed 26 February 2021].
3. Variu, L., 2021. Software Analysis and Verification — Miniproject. [online] Lara.epfl.ch. Available at: <https://lara.epfl.ch/w/_media/sav-report.pdf> [Accessed 26 February 2021].
4. Gheyi, R., Massoni, T., & Borba, P. (2007). A Static Semantics for Alloy and its Impact in Refactorings. Electronic Notes In Theoretical Computer Science, 184, 209-233. doi: 10.1016/j.entcs.2007.03.023 Available at: <https://www.sciencedirect.com/science/article/pii/S1571066107004434>
5. Hongxin Hu, Gail-Joon Ahn. Enabling Verification and Conformance Testing for Access Control Model Available at: <https://cse.buffalo.edu/ hongxinh/papers/sacmat08.pdf>

5 Alloy Code - Dynamic

This is the dynamic code for our Alloy model. It is dynamic because Time and State have been implemented within it. This is the most robust version of the Alloy model, including 7 unique operations, 4 CRUDs, a "State" signature with over 10 operations, fact traces, over 50 total predicates, etc. This model is well constrained and provides ample operations for users at the bank which closely resemble how real world banking policies are leveraged.

```
//Policies For Banks - Jeremy Johnson - March 2021

/*************************************************
                        TABLE OF CONTENTS
This file is organized in way that should make traversing it fairly easy.
        1) First we have my "open" and custome signatures for Time and State. You can see that directly
            ↪ below
        2) Below that we have my unique pre/post/invariant predicates for my system. We created a block
            comment which details them.
        3) Then comes the CRUDs. Each CRUD has a block comment detailing its contents.
        4) Below that is the "State" code. There is a very large block comment that describes everything.
        5) From here on is mostly facts, predicates, assertions that implement Time and constrain the
            ↪ model. Most of this is
           from the initial assignments that has been modified as we progressed.

*************************************************/

open util/ordering [Time] as T
open util/ordering[State] as so
sig Time {}
sig State {
        customerAmount: Int,
        interestRate:Int
}

/*
        ~~~~~~~~~~~~~~~~~~ Unique Pre/Post/Invariants ~~~~~~~~~~~~~~~~~
        -> Here is a comprehensive list of all the unique Pre/Post/Invariant examples we have created
        -> There is one additional Pre/Post example we created in the "States" section, not mentioned
            ↪ here.
```

```
        1) hire new employee -> Found immediately below.
        2) fire employee -> Found immediately below.
        3) Withdraw Money -> Found immediately below.
        4) add customer -> Found in CRUD 1
        5) open account -> Found in CRUD 2
        6) Deposit Check -> Found in CRUD 3
        7) reset Pin on bank card -> Found in CRUD 4
*/

//Unique Pre/Post: Hire an employee to the bank.
pred hire_employee [u: User, u': User, b:MyBank, t,t': Time] {
        //pre condition
        u in Boss
        u' not in u.manager.t
        //post condition
        u' in u.manager.t'
        u' in Employee
        //post condition and frame condition
        u in b.privilegedUser.t and u in b.privilegedUser.t'
        u' not in b.privilegedUser.t and u' in b.privilegedUser.t'
}
pred hire_new_employee {
        all disj u, u':User | all t: Time, b:MyBank | hire_employee[u, u', b, t, T/next[t]]
}
run { hire_new_employee } for 4

//Unique Pre/Post: Fire employee
pred fire_employee [u: User, u': User, b:MyBank, t,t': Time] {
        //pre condition
        u in Boss
        u' in Employee
        u' in u.manager.t
        //post condition
        b.privilegedUser.t' = b.privilegedUser.t - u'
        u' not in u.manager.t'
        //post condition and frame condition
        u in b.privilegedUser.t and u in b.privilegedUser.t'
        u' in b.privilegedUser.t and u' not in b.privilegedUser.t'
}
pred remove_employee {
        all disj u, u':User | all t: Time, b:MyBank | fire_employee[u, u', b, t, T/next[t]]
}
run { remove_employee } for 4


//Unique Pre/Post: Withdraw money from the account
pred make_withdrawal [u: User, b:BankAccount, t,t': Time] {
        //pre condition
        u.account.t.accountValue > 0
        b in u.account.t
        //post condition
        //This is a hard coded withdrawal. Another approach would be to turn this into a function and
            ↪ allow the user to
        //pass a value to the function for the withdrawal.
        u.account.t'.accountValue = u.account.t.accountValue - 3
        u.account.t'.accountValue > 0
        //post condition and frame condition
        b in u.account.t'
}
pred withdraw_money {
        all u:User | all t: Time, b:BankAccount | make_withdrawal[u, b, t, T/next[t]]
}
run { withdraw_money } for 6

/*
        ~~~~~~~~~~~~~~~~~~~~ CRUD 1 ~~~~~~~~~~~~~~~~~~~~~~
        This focuses on the signature "MyBank"
        C - Create a new user in the bank
        R - Return the set of all banks
        U - Update, add a new privileged user to the bank
        D - Delete a user from the bank

        Pre/Post Example: Add a new user to the bank
*/
//CRUD 1 - "C" - Create a new user in the bank.
pred newUser [b:MyBank, u:User, t:Time] {
        init_bank[b]
        init_user[u,t]
        b.regularUser.t = b.regularUser.t + u
}
```

```
pred makeNewRegularUser{
        some b: MyBank, t:Time| all u:User | newUser[b, u, t]
}
run { makeNewRegularUser } for 5

//CRUD 1 - "R" - return the set of all banks
fun return_all_banks[b:MyBank] : set MyBank {
        b
}
run return_all_banks {} for 3

//CRUD 1 - "U" - update privileged user in the bank
pred newPrivilegedUser[b:MyBank, u:User, t:Time] {
        init_bank[b]
        init_user[u,t]
        b.privilegedUser.t = b.privilegedUser.t + u
}
pred makeNewPrivilegedUser{
        some b: MyBank, t:Time| all u:User | u in b.privilegedUser.t implies newPrivilegedUser[b, u, t]
}
run { makeNewPrivilegedUser } for 5

//CRUD 1 - "D" - Delete user from the bank
pred deleteUser[b:MyBank, u:User, t:Time] {
        init_bank[b]
        b.regularUser.t = b.regularUser.t - u
}
pred deleteUserFromBank{
        some b: MyBank, t:Time| some u:User | deleteUser[b, u, t]
}
run { deleteUserFromBank } for 4

//CRUD 1 PRE/POST example
pred add_customer [b: MyBank, u:User, t,t': Time] {
        init_bank[b]
        //pre condition
        b.regularUser.t not in u
        b.privilegedUser.t not in u
        //post conditions
        b.regularUser.t' in u
        //frame condition
        u not in Employee
        u.account.t' in SharedAccount
}
pred new_customer_instance {
        some b: MyBank, t:Time| all u:User | add_customer[b, u, t, T/next[t]]
}
run { new_customer_instance } for 4

/*
        ~~~~~~~~~~~~~~~~~~~~ CRUD 2 ~~~~~~~~~~~~~~~~~~~~~~
        This focuses on the signature "User"
        C - Create a new user
        R - Return the set of all customers / employees
        U - Update user to have a new account
        D - Delete an account from a user.

        Pre/Post: Open a new account for the user.
*/
//CRUD 2 - "C" - Create a new user.
pred createUser[b:MyBank, u:User, t:Time] {
        init_bank[b]
        init_user[u,t]
        b.regularUser.t = b.regularUser.t + u
}
pred createNewUser{
        some b: MyBank, t:Time| all u:User | createUser[b, u, t]
}
run { createNewUser } for 5

//CRUD 2 - "R" - return the set of all users within the bank
fun return_all_users[b:MyBank, t:Time] : set User {
        b.privilegedUser.t + b.regularUser.t
}
run return_all_users {} for 3

//CRUD 2 - "U" - Update user to have a new account.
pred createUserAccount[u:User, a:Account, t:Time] {
        init_user[u,t]
        u.account.t = u.account.t + a
}
```

```
pred createNewUserAccount{
        some u:User, t:Time| one a:Account | createUserAccount[u, a, t]
}
run { createNewUserAccount } for 4

//CRUD 2 - "D" - Delete an account from a user
pred deleteUserAccount[u:User, a:Account, t:Time] {
        init_user[u,t]
        u.account.t = u.account.t - a
}
pred deleteAccountFromUser{
        some u:User, t:Time| one a:Account | deleteUserAccount[u, a, t]
}
run { deleteAccountFromUser } for 4

//CRUD 2 PRE/POST example
pred open_account [u: User, c:checking, t,t': Time] {
        //pre condition
        init_user[u,t]
        init_user[u,t']
        c not in u.account.t
        //post condition
        u.account.t' = u.account.t + c
        //Invariant / frame condition
        c in u.account.t'
}
pred make_new_account {
        one u:User | all c:checking, t: Time | open_account[u, c, t, T/next[t]]
}
run { make_new_account } for 4

/*
        ~~~~~~~~~~~~~~~~~~~~ CRUD 3 ~~~~~~~~~~~~~~~~~~~~~~
        This focuses on the signature "Account"
        C - Create a new account
        R - Return the value of the account
        U - Update, deposit money into the account
        D - Delete account

        Pre/Post: Deposit money into the account from t->t'
*/

//CRUD 3 - "C" - Create a new account
pred createAccount[u:User, a:Account, t:Time] {
        init_user[u,t]
        u.account.t = u.account.t + a
}
pred createNewAccount{
        some u:User, t:Time| one a:Account | createAccount[u, a, t]
}
run { createNewAccount } for 4

//CRUD 3 - "R" - Return the value of the account
fun return_account_value[b:BankAccount] : Int {
        b.accountValue
}
run return_account_value {} for 4

//CRUD 3 - "U" - Update the account with a deposit of money.
pred deposit_money_to_account[b:BankAccount, amount: Int] {
        init_bank_account[b]
        b.accountValue = b.accountValue + amount
}
pred make_deposit{
        some b:BankAccount | deposit_money_to_account[b, 100]
}
run { make_deposit } for 4

//CRUD 3 - "D" - Delete the account
pred removeAccount[b:BankAccount, u:User, t:Time] {
        init_bank_account[b]
        u.account.t = u.account.t - b
}
pred deleteAccount{
        some b:BankAccount, u:User, t:Time | removeAccount[b, u, t]
}
run { deleteAccount } for 4

//CRUD 3 PRE/POST example
pred deposit_money [u: User, b:BankAccount, t,t': Time] {
        //pre condition - bank account is in the user.
```

```
        init_user[u,t]
        init_user[u,t']
        init_bank_account[b]
        u.account.t.accountValue > 0
        //post condition
        u.account.t'.accountValue = u.account.t'.accountValue + 100
        //this is a frame condition.
        b in u.account.t and b in u.account.t'
}
pred make_money_deposit {
        one u:User | all b:BankAccount, t: Time | deposit_money[u, b, t, T/next[t]]
}
run { make_money_deposit } for 4


/*
        ~~~~~~~~~~~~~~~~~~~~ CRUD 4 ~~~~~~~~~~~~~~~~~~~~~~
        This focuses on the signature "BankCard"
        C - Create a new bank card
        R - Return the PIN
        U - Update, change the PIN on the card
        D - Delete the bank card

        Pre/Post: Create a new PIN from t -> t'
*/

//CRUD 4 - "C" - Create a new bank card
pred createBankCard[u:User, b:BankCard, t:Time] {
        init_user[u,t]
        u.card.t = u.card.t + b
}
pred createNewBankCard{
        all u:User, t:Time | one b:BankCard | u in Customer and createBankCard[u, b, t]
}
run { createNewBankCard } for 4

//CRUD 4 - "R" - Return the PIN for the account
fun return_account_value[b:BankCard] : Int {
        b.PIN
}
run return_account_value {} for 4

//CRUD 4 - "U" - Update the PIN on the card to another PIN.
pred update_PIN[b:BankCard, b':BankCard] {
        b.PIN = b'.PIN
}
pred change_pin{
        some b, b':BankCard | update_PIN[b, b']
}
run { change_pin } for 4

//CRUD 4 - "D" - Delete the bank card
pred removeBankCard[b:BankCard, u:User, t:Time] {
        u.card.t = u.card.t - b
}
pred deleteBankCard{
        some b:BankCard, u:User | all t:Time | removeBankCard[b, u, t]
}
run { deleteBankCard } for 7

//CRUD 4 PRE/POST example
pred change_existing_pin [u: User, b:BankCard, b':BankCard, t,t': Time] {
        //pre condition
        init_user[u,t]
        init_user[u,t']
        b'.PIN > 0 //New PIN must be a valid pin.
        b.PIN > 0 //old PIN must also be valid.
        //post condition
        u.card.t'.PIN = b'.PIN
        //Frame Condition - Bank Card must be in the user in both moments in time.
        b in u.card.t and b in u.card.t'
}
pred change_pin_over_time {
        some u:User | all b, b':BankCard, t: Time | u in Customer and change_existing_pin[u, b, b', t, T/
            ↪ next[t]]
}
run { change_pin_over_time } for 5


//Invariants - Check scenarios over all time to hold true. -- refine to thinking about them like "
    ↪ constants"
```

```
//                          Invariants are all displayed in this document in Facts, Assertions, etc.
assert all_customers_regular_users_all_time {
        //read as: For all banks, all users, If the user is a regular user then the user is a civilian.
        all b: MyBank | all u: User, t:Time | u in b.regularUser.t implies u in Customer
}

check all_customers_regular_users_all_time for 5

/*
        ~~~~~~~~~~~~~~~~~~~~~~~~ State Code Below ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        -> My state has an interest rate and a customer amount. We are going to create predicates that
            ↪ work with these concepts.

        -> We have added the following operations:
                1) "init_state" This is initializing states for the customerAmount and interestRate. This
                    ↪  uses "first" so that the first states are guaranteed
                                    to be initialized to my specifications
                2) For "customerAmount", I:
                        -> added an assertion ensuring that customerAmount is always greater than zero.
                        -> Created predicates for adding and removing customers
                        -> created a fact trace that goes through all states besides the last one and
                            ↪ adds customers or removes customers for each cycle.
                3) For "InterestRate", I:
                        -> Created a pre/post/invariant conditional with two states showing me increase
                            ↪ interest rates.
                        -> Created predicates for adding one interest point, remove one interest point,
                            ↪ and clearing interest and setting it to zero.
                        -> Created a fact trace that goes through all states besides the last one and
                            ↪ adds one interest, removes one interest, or
                                clears the interest.

*/
//initial state for my State object. This uses ordering to initialize the first state with values.
pred init_state {
        let firstState = so/first | {firstState.customerAmount = Int[4] and firstState.interestRate = Int
            ↪ [1] }
}
//Created an assertion to add some variety here! Must always have a customer amount greater than zero.
assert zero_or_more_customers {
        init_state
        all s:State | s.customerAmount ≥ 0
}
check zero_or_more_customers for 5
//Takes in two states, the latter state is the previous state customer amount plus 1.
pred add_new_customer(s:State, s':State) {
        s'.customerAmount = s.customerAmount + 1
}
//Takes in two states, this time it removes one customer amount.
pred remove_customer (s:State, s':State) {
        s'.customerAmount = s.customerAmount - 1
}
//We wanted to test calling the predicate directly to ensure it works here. For some States, remove a
    ↪ customer!
pred remove_one_customer {
        init_state
        some s, s' :State | remove_customer[s,s']
}
run remove_one_customer for 6
//A fact trace for customers, iterates over all states besides the last and calls the predicates above.
fact customer_traces {
        init_state
        all s:State - last | let s' = next[s] | add_new_customer[s, s'] or remove_customer[s,s']
}
//This is a predicate trans for the State customerAmount
pred customer_trans[] {
        //for some State s and s', we add a new customer or remove a customer.
        (some s,s':State | add_new_customer[s,s'])
        or (some s,s':State | remove_customer[s,s'])
}
run customer_trans for 5

//this is a pre/post/invariant example we wanted to include. It adds one interest rate from one state to
    ↪ the next.
pred increment_interest[s:State, s':State] {
        //pre conditions
        s.interestRate ≥ 0
        //post condition
        s'.interestRate = s.interestRate + 1
        //frame condition / invariant
        s'.interestRate ≥ 0
}
```

```
//increase the insterest rate by calling the predicate increment_interest
pred increaseInterest {
        some s: State | increment_interest[s, next[s]]
}
run increaseInterest for 6
//decreate the interest rate by one point from one state to the next.
pred decrease_one_interest_point(s:State, s':State) {
        s'.interestRate = s.interestRate - 1
}
//sets the interest rate to zero.. something the fed has basically done now!?!
pred remove_interest(s:State, s':State) {
        s'.interestRate = s.interestRate - s.interestRate //kind of a weird implementation, lets me use
            ↪ both s and s' though!
}
//fact trace for interest, similar to the one implemented for customer amount.
fact interest_traces {
        init_state
        all s:State - last | let s' = next[s] | remove_interest[s, s'] or decrease_one_interest_point[s,s
            ↪ '] or increment_interest[s,s']
}
//This is a predicate trans for the State interestRate
pred interest_trans[] {
        //for some State s and s', we add an interest point, decrease by one point, or clear interest
            ↪ completely.
        (some s,s':State | remove_interest[s,s'])
        or (some s,s':State | decrease_one_interest_point[s,s'])
        or (some s,s':State | increment_interest[s,s'])
}
run interest_trans for 4
//------------------------------------------------- END OF STATES
    ↪ ------------------------------------------------
//Everything below here is signature declarations, facts, assertions, predicates, etc. Mostly from the
    ↪ initial assignments
//edited to work with Time.
//
    ↪ ------------------------------------------------------------------------------------------------------------------------------------------
    ↪
/*
        - With the addition of "Bank" there are many new conditions to consider.
                1) Every User must be connected to a bank.
                2) If a user is a boss or employee, must be connected
*/
one sig MyBank {
        regularUser: set User -> Time,
        privilegedUser: set User -> Time,
}
/*
        ---- Initial State Predicate ----
        -> This is the initial state for the bank sig
*/
pred init_bank (b:MyBank) {
        #b= 1
}


fact "All␣Users␣are␣connected␣to␣a␣bank␣with␣EITHER␣privilege␣or␣regular␣status"{
        //read as: For all user U, there is one or more bank such that U is a regular user or privileged
            ↪ user.
        all u:User, t:Time | some b:MyBank | u in b.regularUser.t or u in b.privilegedUser.t
}

fact "All␣Customers␣do␣not␣have␣the␣privilege␣connection." {
        //read as: for all customers C, there is no bank such that C has privileged user.
        all c:Customer, t:Time | no bank:MyBank | c in bank.privilegedUser.t
}

fact "All␣Employees␣and␣Bosses␣have␣privilege␣trait␣from␣bank" {
        //read as: For all employee E, there is no bank such that E is a regular user
        all e:Employee, t:Time | no bank:MyBank | e in bank.regularUser.t
        //read as: For all boss B, there is no bank such that B is a regular user
        all b:Boss, t:Time | no bank:MyBank | b in bank.regularUser.t
}

/*
        - Users are members of the bank.
*/
sig User {
        account: set Account -> Time
}

/*
```

```
        - The SharedAccount state can exist between two users.
*/

fact "Only␣Customers␣can␣share␣accounts" {
        //read as: For all users, if the user has a SharedAccount then the user is a customer.
        all u:User, t:Time | u.account.t in SharedAccount implies u in Customer
}

fact "no␣account␣belongs␣to␣customers␣and␣employees" {
        all u:User, a:Account, t:Time | u.account.t in a and u in Customer iff u not in Employee and u
            ↪ not in Boss
}


/*
        - The Manager state can exist and can be connected to one user who is the employee.
*/
sig Boss in User {
        manager: set Employee -> Time,
}
sig Employee in User {}
some sig Customer in User {
        card: set BankCard -> Time,
}
sig BankCard {
        PIN: Int
}
{
        PIN > 0
}

/*
        ---- Initial State Predicate ----
        -> This is the initial state for the user sig
*/
pred init_user (u:User, t:Time) {
        //Employes >= Boss ensures there are no extra bosses at the bank. Also PINs need to be > 0.
        #Employee ≥ #Boss
        u.card.t.PIN > 0
}


fact "For␣all␣users,␣you␣can␣only␣have␣one␣bank␣account␣within␣a␣bank.␣This␣doesn't␣prevent␣multiple␣
    ↪ checking/savings␣accounts."{
        //read as: for all users u, all disjoin bank accounts, user having one account implies user doesn
            ↪ 't have another account.
        all u:User, t:Time | all disj a,a':BankAccount | a in u.account.t implies a' not in u.account.t
}

fact "no␣BankCard␣not␣connected␣to␣a␣user" {
        //read as: for all users u, there is no ID i such that the user doesn't have an ID.
        all u:User, t:Time | no b:BankCard | u.card.t not in b
}


fact "No␣employees␣or␣bosses␣are␣customers" {
        //read as: For all employee E, there is no customer such that the civilian is an employee.
        all e:Employee | no c: Customer | c in e
        //read as: For all bosses B, there is no customer such that the civilian is a boss.
        all b:Boss | no c: Customer | c in b
}

fact "User␣can␣not␣be␣a␣Boss␣and␣Employee␣at␣the␣same␣time.␣User␣can␣also␣be␣a␣Customer."{
        //read as: For all Users U, U can be an employee, boss, or civilian, but not more than one at
            ↪ once.
        all u: User | (u in Employee and u not in Boss) or (u in Boss and u not in Employee) or (u in
            ↪ Customer)
}

fact "If␣you␣share␣an␣account␣then␣the␣account␣is␣shared␣back." {
        //read as: All shared accounts are shared both ways.
        shared = ~shared
}

/*
        - This is the Account signature, there are checking and savings accounts.
*/
abstract sig Account {}

/*
        - There are checkings and savings accounts that are valid instantiations of "Account".
```

```
                - Each account can connect to a user with a "primary" or "secondary" connection.
*/

sig BankAccount extends Account {
        accountValue: Int
}
sig checking, saving in Account {}
sig SharedAccount in Account {
        shared: one Account
}
/*
        ---- Initial State Predicate ----
        -> This is the initial state desired for the Bank Account sig
*/
pred init_bank_account (b:BankAccount) {
        //Account balance > 0 and no bank account shares an account with itself.
        b.accountValue > 0
        b not in b.shared
}


fact "Accounts are only shared if they belong to a customer." {
        //read as: For all users in all time, the account is in shared account implies the user is a
            ↪ customer.
        all a:Account, u:User, t:Time | a in SharedAccount implies a in u.account.t and u in Customer
}

fact "No account can share an account with itself" {
        //read as: For all users U, U is not sharing account with itself.
        all b:BankAccount | b not in b.shared
}


fact "All savings and checkings accounts must be in a User's account" {
        //read as: There does not exist a saving/checking account for all users such that the saving/
            ↪ checking account is not in the users account
        //This enforces that there shall be no accounts with no users!
        no b: BankAccount | all u:User, t:Time | b not in u.account.t
        //no s: saving | all u:User, t:Time | s not in u.account.t
}

/*
        ASSERTIONS to make:
        1) Accounts do not share themselves
        2) All users are connected to the bank with regularUser or PrivilegedUer
        3) All customers are regular users.

*/
assert accounts_no_self_share {
        //read as: for all bank accounts b, b is not in its own shared account.
        all b:BankAccount | b not in b.shared
}
assert no_users_with_dual_statuses {
        //read as: For all banks, and for all users, each user has either regular user OR privileged user
            ↪ , but
        //                        not both. Having both would violate a basic security policy.
        all b: MyBank | all u: User, t:Time | u in b.regularUser.t implies u not in b.privilegedUser.t
}
assert all_customers_are_regular_users {
        //read as: For all banks, all users, If the user is a regular user then the user is a customer.
        all b: MyBank | all u: User, t:Time | u in b.regularUser.t implies u in Customer
}

check accounts_no_self_share for 5
check no_users_with_dual_statuses for 5
check all_customers_are_regular_users for 5

/*
predicates:
        1) customers always have one checking account
        2) bosses and employees do not share accounts
        3) every employee has a boss who is their manager
*/
pred checking_in_customer[u:User, c: checking, t:Time] {
        init_user[u,t]
        //read as: checking account provided is in the users account.
        c in u.account.t
}
pred bank_accounts_no_self_share[b: BankAccount] {
        init_bank_account[b]
        //read as:  for all bank accounts b, b is not in its own shared account.
```

```
        b not in b.shared
}
pred boss_are_managers_for_employees[u:User, u':User, t:Time] {
        //read as: for the provided boss and employee, the boss is the employees manager.
        init_user[u,t]
        init_user[u',t]
        //read as: for the provided boss and employee, the boss is the employees manager.
        u in Employee and u' in Boss implies u in u'.manager.t
}

run checking_in_customer {} for 5
run bank_accounts_no_self_share {} for 5
run boss_are_managers_for_employees {} for 5

/*
Functions
        1) Returns a set of the two provided users accounts
        2) Returns the set of employees that two bosses are managing.
        3) Returns the set of all regular users inside of a bank.
*/
fun accounts_belonging_to_two_users[u, u': User, t:Time] : set Account {
        //read as: The function accepts two users accounts, returns a set of accounts. The functions
        //                      combines both the users accounts and returns the union of both.
        u.account.t + u'.account.t
}

fun set_of_employees_between_two_bosses[b:Boss, b':Boss, t:Time] : set Employee {
        //read as: The function accepts two bosses and returns the set of employees belonging to both.
        b.manager.t + b'.manager.t
}

fun all_regular_banks_accounts[b: MyBank, t: Time] : set User {
        //read as: The function accepts a bank and returns the set of users that belong to the bank.
        b.regularUser.t
}

run accounts_belonging_to_two_users {} for 5
run set_of_employees_between_two_bosses {} for 5
run all_regular_banks_accounts {} for 3

run {} for 3
```

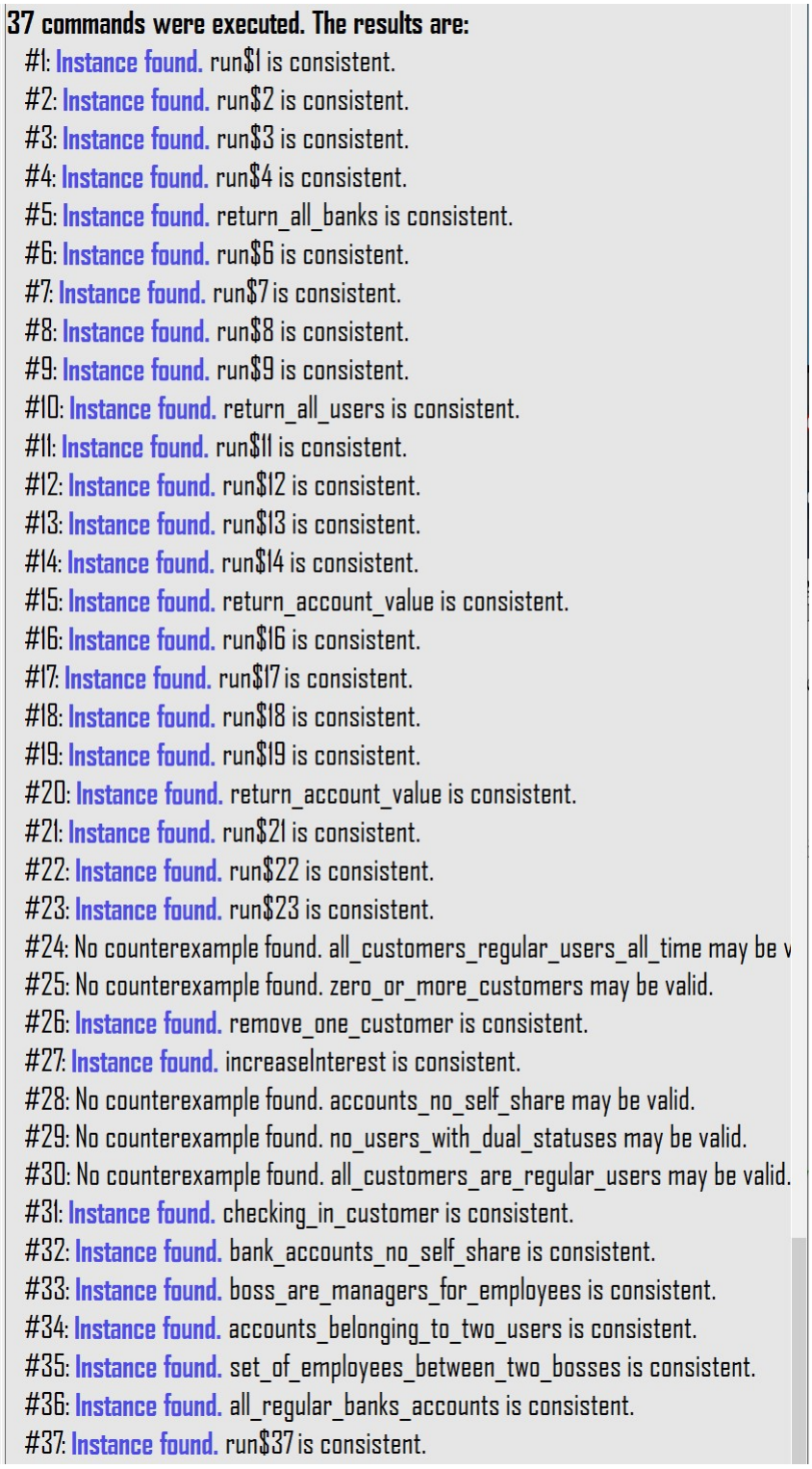

Figure 11: Clicking execute all in Alloy, code compiles!

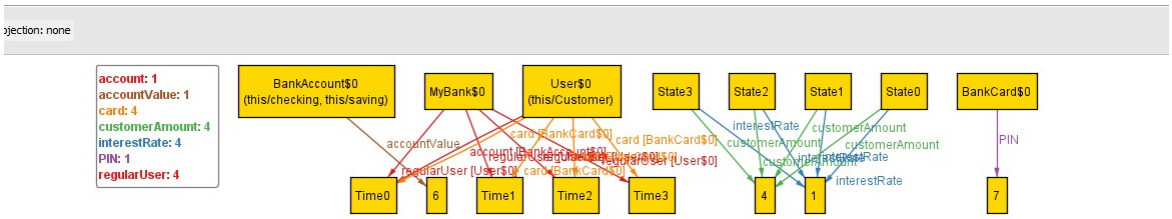

Figure 12: Alloy Model for Hiring Employee Pre/Post image

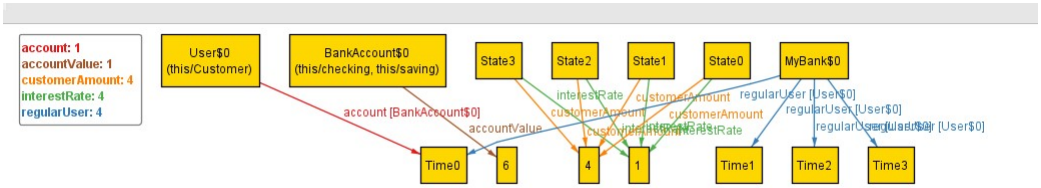

Figure 13: Alloy Model for Firing Employee Pre/Post image

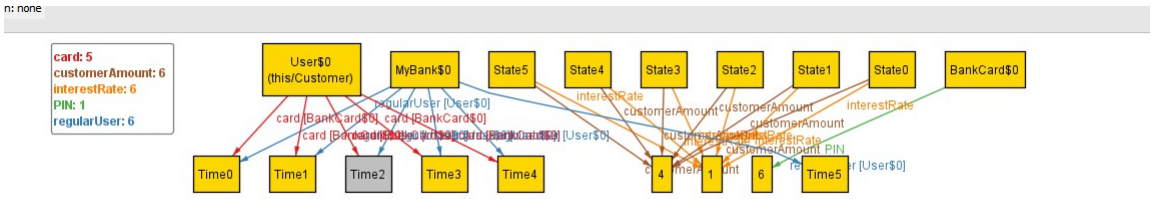

Figure 14: Alloy Model withdraw money Pre/Post image

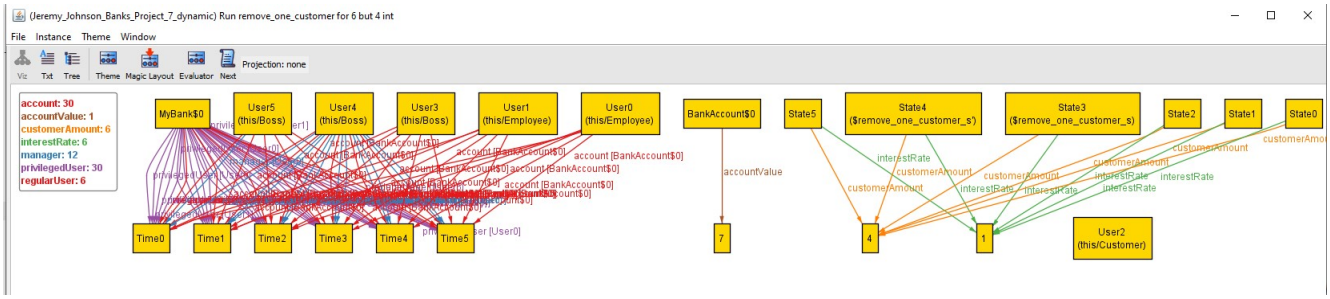

Figure 15: Alloy Model remove one customer State image

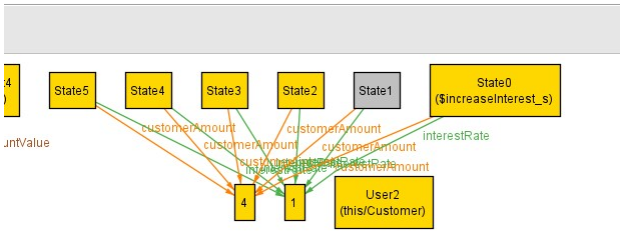

Figure 16: Alloy Model Increase interest rate State image

6 Alloy Code - Static

This is the static version of our program before the addition of Time and State. This is significantly trimmed down from the dynamic model, it does not contain the majority of the operations created in the dynamic version that implements Time and State. This version of the model helped with testing small changes before including them in the dynamic model and was a significant milestone in the progression of our model in this class.

```
//Policies For Banks - Jeremy Johnson - February 2021
/*
        - With the addition of "Bank" there are many new conditions to consider.
                1) Every User must be connected to a bank.
                2) If a user is a boss or employee, must be connected
*/
one sig MyBank {
        regularUser: set User,
        privilegedUser: set User
}
/*
        ---- Initial State Predicate ----
        -> This is the initial state for the bank sig
*/
pred init_bank (b:MyBank) {
        #b= 1
}


fact "All␣Users␣are␣connected␣to␣a␣bank␣with␣EITHER␣privilege␣or␣regular␣status"{
        //read as: For all user U, there is one or more bank such that U is a regular user or privileged
            ↪ user.
        all u:User | some b:MyBank | u in b.regularUser or u in b.privilegedUser
}

fact "All␣Customers␣do␣not␣have␣the␣privilege␣connection." {
        //read as: for all customers C, there is no bank such that C has privileged user.
        all c:Customer | no bank:MyBank | c in bank.privilegedUser
}

fact "All␣Employees␣and␣Bosses␣have␣privilege␣trait␣from␣bank" {
        //read as: For all employee E, there is no bank such that E is a regular user
        all e:Employee | no bank:MyBank | e in bank.regularUser
        //read as: For all boss B, there is no bank such that B is a regular user
        all b:Boss | no bank:MyBank | b in bank.regularUser
}

/*
        - Users are members of the bank.
*/
sig User {
        account: set Account
}

/*
        - The SharedAccount state can exist between two users.
*/

fact "Only␣Customers␣can␣share␣accounts" {
        //read as: For all users, if the user has a SharedAccount then the user is a customer.
        all u:User | u.account in SharedAccount implies u in Customer
}
```

```
fact "no account belongs to customers and employees" {
        all u:User, a:Account | u.account in a and u in Customer iff u not in Employee and u not in Boss
}


/*
        - The Manager state can exist and can be connected to one user who is the employee.
*/
some sig Boss in User {
        manager: set Employee,
}
some sig Employee in User {}
some sig Customer in User {
        card: one BankCard,
}
sig BankCard {
        PIN: Int
}
{
        PIN > 0
}

/*
        ---- Initial State Predicate ----
        -> This is the initial state for the user sig
*/
pred init_user (u:User) {
        //u in Employee implies u in manager
        #Employee ≥ #Boss
        #Customer ≥ 0
}

fact "no BankCard not connected to a user" {
        //read as: for all users u, there is no ID i such that the user doesn't have an ID.
        all u:User | no b:BankCard | u.card not in b
}



fact "For all users, you can only have one bank account within a bank. This doesn't prevent multiple
    ↪ checking/savings accounts."{
        //read as: for all users u, all disjoin bank accounts, user having one account implies user doesn
            ↪ 't have another account.
        all u:User | all disj a,a':BankAccount | a in u.account implies a' not in u.account
}




fact "No employees or bosses are customers" {
        //read as: For all employee E, there is no customer such that the civilian is an employee.
        all e:Employee | no c: Customer | c in e
        //read as: For all bosses B, there is no customer such that the civilian is a boss.
        all b:Boss | no c: Customer | c in b
}

fact "User can not be a Boss and Employee at the same time. User can also be a Customer."{
        //read as: For all Users U, U can be an employee, boss, or civilian, but not more than one at
            ↪ once.
        all u: User | (u in Employee and u not in Boss) or (u in Boss and u not in Employee) or (u in
            ↪ Customer)
}

fact "If you share an account then the account is shared back." {
        //read as: All shared accounts are shared both ways.
        shared = ~shared
}

/*
        - This is the Account signature, there are checking and savings accounts.
*/
abstract sig Account {}

/*
        - There are checkings and savings accounts that are valid instantiations of "Account".
                - Each account can connect to a user with a "primary" or "secondary" connection.
*/

sig BankAccount extends Account {
```

```
        accountValue: Int
}

sig checking, saving in Account {}

sig SharedAccount in Account {
        shared: one Account
}

/*
        ---- Initial State Predicate ----
        -> This is the initial state desired for the Bank Account sig
*/
pred init_bank_account (b:BankAccount) {
        //Account balance > 0 and no bank account shares an account with itself.
        b.accountValue > 0
        b not in b.shared
}


fact "Accounts are only shared if they belong to a customer." {
        //read as: For all users in all time, the account is in shared account implies the user is a
            ↪ customer.
        all a:Account, u:User | a in SharedAccount implies a in u.account and u in Customer
}

fact "No account can share an account with itself" {
        //read as: For all users U, U is not sharing account with itself.
        all b:BankAccount | b not in b.shared
}


fact "All savings and checkings accounts must be in a User's account" {
        //read as: There does not exist a saving/checking account for all users such that the saving/
            ↪ checking account is not in the users account
        //This enforces that there shall be no accounts with no users!
        no b: BankAccount | all u:User | b not in u.account

}

/*
        ASSERTIONS to make:
        1) Accounts do not share themselves
        2) All users are connected to the bank with regularUser or PrivilegedUer
        3) All customers are regular users.

*/
assert accounts_no_self_share {
        //read as: for all bank accounts b, b is not in its own shared account.
        all b:BankAccount | b not in b.shared
}
assert no_users_with_dual_statuses {
        //read as: For all banks, and for all users, each user has either regular user OR privileged user
            ↪ , but
        //                        not both. Having both would violate a basic security policy.
        all b: MyBank | all u: User | u in b.regularUser implies u not in b.privilegedUser// not in u
}
assert all_customers_are_regular_users {
        //read as: For all banks, all users, If the user is a regular user then the user is a customer.
        all b: MyBank | all u: User | u in b.regularUser implies u in Customer
}

check accounts_no_self_share for 5
check no_users_with_dual_statuses for 5
check all_customers_are_regular_users for 5

/*
predicates:
        1) customers always have one checking account
        2) bosses and employees do not share accounts
        3) every employee has a boss who is their manager
*/

// init_user (u:User) {
// init_bank_account (b:BankAccount) {
pred checking_in_customer[u:User, c: checking] {
        init_user[u]
        //read as: checking account provided is in the users account.
        c in u.account
}
pred bank_accounts_no_self_share[b: BankAccount] {
```

```
        init_bank_account[b]
        //read as:  for all bank accounts b, b is not in its own shared account.
        b not in b.shared
}

pred boss_are_managers_for_employees[u:User, u':User] {
        init_user[u]
        init_user[u']
        //read as: for the provided boss and employee, the boss is the employees manager.
        u in Employee and u' in Boss implies u in u'.manager
}

run checking_in_customer {} for 5
run bank_accounts_no_self_share {} for 5
run boss_are_managers_for_employees {} for 5

/*
Functions
        1) Returns a set of the two provided users accounts
        2) Returns the set of employees that two bosses are managing.
        3) Returns the set of all regular users inside of a bank.
*/
fun accounts_belonging_to_two_users[u, u': User] : set Account {
        //read as: The function accepts two users accounts, returns a set of accounts. The functions
        //                      combines both the users accounts and returns the union of both.
        u.account + u'.account
}

fun set_of_employees_between_two_bosses[b:Boss, b':Boss] : set Employee {
        //read as: The function accepts two bosses and returns the set of employees belonging to both.
        b.manager + b'.manager
}

fun all_regular_banks_accounts[b: MyBank] : set User {
        //read as: The function accepts a bank and returns the set of users that belong to the bank.
        b.regularUser
}

run accounts_belonging_to_two_users {} for 5
run set_of_employees_between_two_bosses {} for 5
run all_regular_banks_accounts {} for 3

run {} for 8
```

```
10 commands were executed. The results are:
  #1: No counterexample found. accounts_no_self_share may be valid.
  #2: No counterexample found. no_users_with_dual_statuses may be valid
  #3: No counterexample found. all_customers_are_regular_users may be
  #4: Instance found. checking_in_customer is consistent.
  #5: Instance found. bank_accounts_no_self_share is consistent.
  #6: Instance found. boss_are_managers_for_employees is consistent.
  #7: Instance found. accounts_belonging_to_two_users is consistent.
  #8: Instance found. set_of_employees_between_two_bosses is consisten
  #9: Instance found. all_regular_banks_accounts is consistent.
  #10: Instance found. run$10 is consistent.
```

Figure 17: Alloy Code Static Compiles

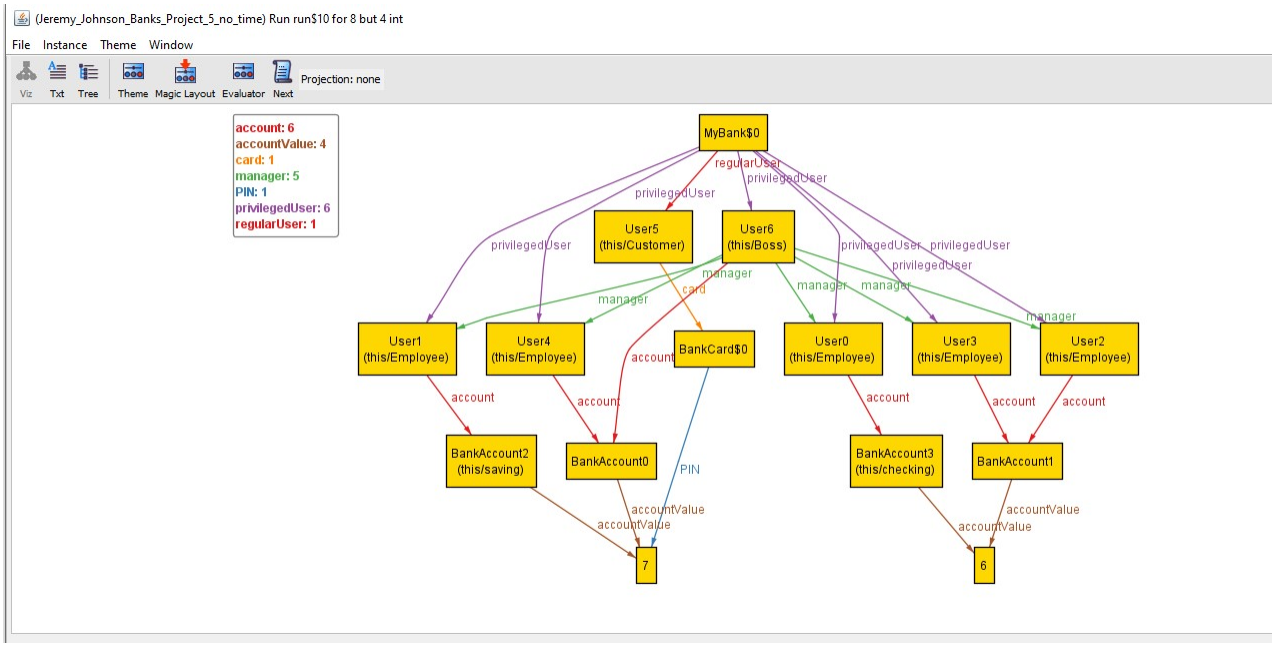

Figure 18: Alloy Code Static Model Example