The Report committee for Kelly Thomas Howell

Certifies that this is the approved version of the following report

Application of Techniques to Test Software Designs Against Requirements

Approved by

Supervising Committee:

Supervisor: _____

Christine Julien

_____

Tom Graser

Application of Techniques to Test Software Designs Against Requirements

by

Kelly Thomas Howell, BS

Master Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2009

Dedication

To Lana and Kathryn Ruth

In Memory

Ruth Wathen

Abstract

Application of Techniques to Test Software Designs Against Requirements

by

Kelly Thomas Howell, M.S.E.

The University of Texas at Austin, 2009

Supervisor: Christine Julien

Engineers in diverse fields are able to model their design and experiment with that design to determine how it responds to the environment and how it satisfies the requirements. Design tools for software engineering have become standardized and matured to allow for formal definition of software design. This paper tests the current state of design documentation to determine the quality of design testing available at the early stage of software design.

Table of Contents

List of Tables ix

List of Figures x

List of Figures x

APPLICATION OF TECHNIQUES TO TEST SOFTWARE DESIGNS AGAINST REQUIREMENTS **1**

Abstract 1

Modeling Software Requirements 5

Modeling Maturity 6

OCL DESCRIPTION **7**

OCL's relationship with UML 9

Applicability of OCL to MDA 9

CASE STUDY OF APPLICATION **12**

Problem Description 12

Use Cases 13

OCL description of the UML Structure 19

Comments about the Code 24

Action languages 32

Examining the Application 32

Testing the Software Model 38

Test Cases 38

Action Script for Use Cases 39

Conclusions 44

Appendix A – Action Script for Deposit 47

Appendix B – Action Script for Transfer 47

Appendix C – Action Script to Create System State 1 48

Appendix D – Action Script to Create System State 2 52

Appendix E - Object Diagram from Use after Execution of Test Case 53

Sequence Diagram from USE after Execution 54

Appendix F – Unit Test Script Validation Ouput from USE Tool 55

References 61

Vita 63

List of Tables

Table: Test Execution Script..38

List of Figures

Deposit Money Use Case Diagram: 14
Withdraw Money Use Case Diagram: 14
Allocate Money Use Case Diagram: 15
Set Goal Use Case Diagram: 16
Transfer Funds Use Case Diagram: 16
Example System UML Diagram 17
Example Application Sequence Diagram: Deposit Money 30
Sequence Diagram: Transfer Funds 31
OBJECT DIAGRAM SYSTEM STATE 1 33
Object Diagram System State 2: 35

APPLICATION OF TECHNIQUES TO TEST SOFTWARE DESIGNS AGAINST REQUIREMENTS

Abstract

Engineers in the fields of construction, aerospace, nautical engineering, automotive engineering, and many other fields make use of models to test their design against the requirements and environmental conditions. These models are scale versions of the complete object or vehicle, and the designs are refined based upon the results of the model against testing. In contrast, software engineers commonly perform a proof of concept to test their architecture. This proof of concept is generally an implementation of a small subset of the system requirements, developed to determine if the application is technically possible. But these proofs of concepts are not a reliable test of requirements due to the nature of the small subset.

Can software engineers use more formal methods to define their software design and formally experiment with the design of the whole system by experimenting with all of the system requirements? Can software engineers use Unified Modeling Language (UML) models to formally test software design against the use cases? Such experimentation will utilize models of the system to formally test the whole system against all of the stated requirements and lead to earlier detection of costly design defects.

Introduction

Engineers in the fields of construction, aerospace, nautical engineering, automotive engineering, and many other fields make use of models to test their design against the requirements and environmental conditions. These models are scale versions

of the complete object or vehicle, and the designs are refined based upon the results of the model against testing. In contrast software engineers commonly perform a proof of concept to test their architecture. These proofs of concept are short and incomplete realizations of the software or software architecture to demonstrate in principle that the software is technically capable of the proposed solution. These techniques are by definition small and inadequately define the full requirements for the full system. The proof of concept is focused on a single problem or a single characteristic associated with the software requirement.

Can software engineers use more formal methods to define their software design and formally experiment with the design of the whole system by testing with all of the system requirements as documented in the system use cases? Such experimentation will utilize models of the system to formally test the whole system against all of the stated requirements and lead to earlier detection of costly design defects. Can software engineers use more formal methods to describe the software architectures and environments in which the computer software will operate?

Object oriented system languages combine the definition of state and behavior. The Universal Modeling Language provides a standard way to describe the many characteristics of these objects. The Object Constraint Language, defined as part of the Unified Modeling Language (UML) 2.0 standard, provides a means to add formal mathematical logic to these objects.

This paper examines a small sample set of software requirements that will be described in traditional text, UML, and use-case documentation. These requirements will be transformed to a more formal language, Object Constraint Language (OCL), and Action Scripts, in combination with UML [3, 10]. These artifacts will be used to test the

functional requirements and determine if formal documentation tools allow for accurate unit testing performed on the software model.

The USE specification tool developed by Martin Gogolla et al. at the University of Bremen, in Bremen Germany, was used for the specific implementation tool of OCL and UML.

Software Modeling Background

Computer software is a series of levels of abstraction building upon the binary calculator that lies as the bottom as the physical reality implemented in a silicon chip. The underlying calculator is a powerful tool to implement logic, but to abstract this useful behavior for developers, programming languages were developed to provide abstraction models for higher levels of logic. Applications are developed using these higher level development languages to solve problems and meet the specified needs of end users. The application software and development software make use of models to represent and manipulate information.

Model analysis is often associated with Object Oriented (OO) languages due to the encapsulation of data and behavior in an abstraction named an object. However, model analysis can also be performed with Entity Relationship Diagrams (ERD) and data flow charts [3]. "A model is an abstraction of a system from a particular viewpoint. It describes the system or entity at the chosen level of precision and viewpoint." The success of the abstraction methodology is demonstrated by the complexity of modern systems, rather than an improvement in development speed or reduced development cost. Models help developers cope with larger and more complex systems, and having a good model has made the process of writing code easier and more straightforward [7, 9].

Models can also be created to describe the domains for which software is developed. Software modeling attempts to identify the elements of the domain, their associations, relationships, connections, and behavior and represents these elements as software packages, systems, classes, or services that can be implemented in a software system. No one view of any system is complete as a stand alone model. Many different models are used in conjunction to provide a complete description of the system [7, 11].

The Unified Modeling Language (UML) has become the standard for documenting software models, especially for object oriented systems [2, 3]. The system's state and data are represented with UML diagrams. UML allows for system behavior to be documented with views known as state machines, activity diagrams, and use cases. Many software requirements are represented using a series of use case views to indicate what functionality the software will perform for given sets of users. These uses cases represent the functional requirements for the software. Other important requirements are captured as non-functional requirements that relate the features of the software such as performance, modifiability, availability, security, and testability. For this paper we will focus on modeling and testing the functional requirements.

Use cases are frequently used to capture initial requirements for a system by producing a use case for each functional thread. The elements in these uses cases represent the system itself, and each interaction involves an actor in the system's environment.

From The Unified Modeling Language Reference Manual:

> The use case view models the functionality of a subject (such as a system) as perceived by outside agents, called actors, (which) interact with the subject from a particular viewpoint. A use case is a unit of functionality expressed as a transaction among actors and the subject. The purpose of the use case view is to list the actors and show which actors participate in each use case. [7]

Use cases often include variations of the main sequence of interactions in order to document exceptional conditions. Uses cases should be kept simple; all interactions in a use case are between the subject and the actors in its environment.

Modeling Software Requirements

Developing models from software requirements and the application domain elevates the level of abstraction. This increased abstraction has several benefits:

Portability: The models are associated with the business domain, which increases application and re-use. The level of abstraction is less likely to change, and modeling at this level isolates the model from details that have a higher rate of change. Modeling brings portability and platform independence because the model is platform and technology independent.

Productivity: Modeling increases productivity by enabling developers, designers, and system administrators to use languages and concepts with which they are more comfortable.

Cross-Platform Interoperability: Modeling allows for a description of the system that is stated in a medium that is not dependent upon a technology, framework, or computer platform.

Easier Documentation: Modeling improves the documentation of the software. The elements in the model can be targeted for specific perspectives using concepts understood by the target audience. A model may have many perspectives for the many different stakeholders in the system.

MODELING MATURITY

Organizations in the software community employ various levels of sophistication in modeling and requirements management. In Jos Warner's book on the Object Constraint Language [10], he draws a correlation between the CMM level of the organization and the maturity of the requirements developed by the organization. He proposes that the greater the use of models the greater the rigor and less ambiguity in the requirements.

The trend is that as more time and effort are spent documenting the software designs in a model, the greater the effectiveness of the organization to develop software. The use of models provides abstractions that increase the effectives of the software designs [7]. To add to the benefit of software design models, organizations can use these models to test against their requirements, and thereby utilize the model to determine how the design will function when given the stimulus specified by the use cases.

With the greater level of detail and the greater level of formality, these detailed designs will be the artifacts used in our analysis against the system requirements. The modeling language must be unambiguous to allow for systematic testing, much like the mathematical models used in the analysis of a physical process. The model provides a series of rules and conditions necessary for valid states and transitions.

This paper will explore the process and techniques associated with modeling maturity level four by applying them to a set of real world requirements using the elements of UML as the unambiguous language.

OCL DESCRIPTION

The Object Constraint Language (OCL) is a textual language with which the modeler can improve the precision of UML descriptions and is defined as a part of the UML standard for object-oriented analysis and design. OCL is a declarative language: the expressions in the model are lifted fully into the realm of pure modeling, without regard for the in-depth details of the implementation and the implementation language. An expression specifies values at a high level of abstraction and remains precise.

In the second edition of UML, the Object Constraint Language was expanded to express constraints, guard conditions, queries, referencing values, conditions, actions, and business rules. The Object Management Group (OMG) maintains the standard definition for the language.

The OCL descriptions are defined in the context of the UML diagrams of the defined software system. The OCL description offers a precise specification language that enables the realization of a model. The UML model combined with the OCL enables precision and an unambiguous model definition that can be utilized by tools for a more formal analysis.

> Any model must be integrated, clear, and consistent. It must be clear how entities shown in one diagram relate to entities in other visible parts of the model. OCL expressions are often not shown in any diagrams, but they are still part of the model. They are present in the underlying repository. The relation between OCL expressions and entities must be clear. The link between an entity in a UML diagram and OCL expression is called the context of an OCL expression. [10]

The documentation of behavior refers to the structure, and uses the structural elements of a view are an essential part of the language. The formal UML diagrams document the structural elements along with the relationships.

A description of a static state is not sufficient to describe a system. As stated by Clements, et al:

> Any language that supports documenting system behavior must include constructs for describing sequences of interaction [7].

The current UML modeling has two means for documenting sequences of interaction: sequence diagrams and state machine diagrams [2, 3, 6]. However, these tools do not have a formal language that a machine can interpret. To fill this void, action languages have been defined by the Model Driven Architecture community [9, 18].

The conditional OCL expressions represent the constraints, and the Action Languages represent stimuli from the uses cases and system activity. The actions of the Action Languages will provide the stimulus from the user interaction with the software design, and the actions required to transition the system between different states described in the system state diagrams.

Execution of testing: Traces

Given this level of formal description of the system design, we can experiment with the system requirements. The software design as expressed in the modeling language will act as the controlled environment with well defined properties that must hold. The user's requirements are the stimulus to change the state of the environment. The user's requirements, represented as use cases are treated as traces.

> Traces are sequences of activities or interactions that describe the system's response to a specific stimulus when the system is in a specific state. These sequences document the trace of activity through a system described in terms of its structural elements and their interactions [7].

These traces are taken from the use cases presented in the system requirements. For formal exercising of the system model and its reaction, the use cases are translated to action language scripts.

OCL'S RELATIONSHIP WITH UML

The Unified Modeling Language is a series of standardized symbols, shapes, connectors, and text that have a defined meaning to communicate the design of software components and their relationships to other components. These symbols provide meaningful representation to readers but do not provide the unambiguous language necessary for a formal analysis of a software design. In order for UML to provide a description of an environment with rigorous rules, more detail and precision is needed. The Object Constraint Language (OCL) provides this additional level of detail.

OCL has been a part of the OMG standard for UML since version 1.1. OCL was originally developed at IBM in 1995 as a business engineering language and was formally adopted as a formal specification language for UML [14]. The following is a definition of the Object Constraint Language from the UML reference manual:

> The object constraint language is a text language for writing navigation expressions, Boolean expressions, and other queries. It may be used to construct expressions for constraints, guard conditions, actions, preconditions and post-conditions, assertions and other kind of UML expressions. The OCL is defined by the OMG as a companion specification for UML [3].

OCL is used in the UML semantics to document precise definition to determine the well-formed rules for models described in UML.

APPLICABILITY OF OCL TO MDA

Model driven architecture (MDA) is relatively a new concept in the field of computer science. MDA is a framework for software development defined by the Object

Management Group. The key to MDA is the importance of the software model in the development process [8]. The approach to development focuses the majority of effort on the initial design process and the model for the software. The development artifacts of the MDA approach are formal software models that adhere to the UML standard.

In an MDA environment, these platform-independent models are then transformed into platform-specific models. This transformation is defined by developers using knowledge of the specific target domain. The underlying concept is that this transformation definition is a large effort that requires extensive knowledge of the specific target platform. The advantage of the methodology is that this transformation only needs to be defined once. After the initial definition, the transformation can be applied to any platform-independent model.

The object constraint language (OCL) as a query and expression language for UML is an integral part of the scope for MDA.

> Using the combination of UML with OCL to build platform-independent models allows for consistent and precise models. [10]
>
> The strong structural aspects of UML can be utilized and made fully complete and consistent. Query operations can be defined completely by writing the body of the operations as an OCL expression. The dynamics of the system can be written in pre and post conditions that are defined on the operations. [10]

Current State of MDA

The large vision of Model Driven Architecture is not a reality given the current state of technology, and there are some clear divisions about the future of MDA. Model Driven Architecture today is a series of best practices and manual processes that developers can adopt in the hopes that future progress will be gained in small steps until all of the manual portions of the tasks have been automated [8].

Critics of MDA believe that the various platforms available differ too much and provide divergent features that make MDA impossible to achieve. Some critics believe that the emergence of service-oriented architectures and web services have negated the need for the MDA goal of software generated from models [13, 15].

These critics state that MDA is "predicated on the assumption that the implementation platform doesn't matter." They argue that the features and details of the implementation environment, which include the physical environment and the implementation language, are too diverse and as a result unable to hide by way of abstraction. [18]

These same critics argue that service-oriented architectures provide documented interfaces that allow for high-level abstraction of functionality by way of an exposed contract of services which hide the implementation details, such as physical environment and implementation language.

For this investigation no implementation code was generated, and these criticisms did not impact the exploration of testing against more formal models. We made use of some tools associated with the MDA community but did not take the next step to the development of platform-specific models and platform-specific implementations.

CASE STUDY OF APPLICATION

To explore these ideas concerning formal software analysis of a system model based upon a UML description, we have implemented the techniques on a simple real world example application that has enough complexity to test the foundations of model-driven testing.

The business requirements for this application are represented with textual descriptions and use case diagrams collected from the system stakeholders. The system design is modeled using UML and OCL textual representation. The use cases or traces are used to provide the stimulus necessary to cause the state changes described by the system requirements represented as predetermined test cases. The model will define the valid environment in which these state changes will take place.

PROBLEM DESCRIPTION

Background

Small businesses have multiple types of expenses associated with day-to-day operations and larger expenses associated with payroll and rent. The business often have the need to maintain many different funds in order to keep these expenses separated, but financial institutions offer the business only a small number of actual bank accounts.

To reconcile the differences, some financial institutions offer fund services to allow for flexibility. The business can have a handful of accounts but may allocate that money to multiple funds from which money can be withdrawn. This fund system allows for the business to separate and budget their assets to fit the business needs, yet it allows for the financial institution to maintain a single account.

The user can set goals for each fund to specify how much money is needed to meet necessary expenses, such as rent. The rent fund is associated with the business' primary account, and the fund has an associated goal that is equal to the amount of the monthly rent. As deposits are made into the account, a percentage of each deposit is associated with the rent fund, and the user can track if this funding is sufficient to meet the goal of paying the rent at the end of the month.

This system requires that all money is associated to a fund. Therefore each account must have at least one fund; a default "general" fund will be associated with each account.

USE CASES

Use Case:	Deposit Money
Perspective:	Administration
Brief Description:	New money is added to an account; this activity automatically creates a deposit transaction
Business Goals:	All money is initially added at the account level and placed into the general fund for distribution to the other funds.
Actors:	Authorized User

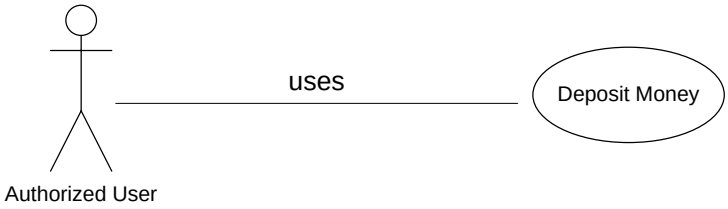
Preconditions:

1. The user must be authorized for the transaction.
2. An account must exist prior to adding money.
3. The amount of the deposit must be greater than zero.
4. The account must have an active status.

Post-conditions:

1. The amount of the deposit is added to the general fund.
2. The account balance is the sum of the individual fund balance for each fund associated with the account.
3. An account level transaction records the deposit.
4. A fund level transaction records the deposit.

Deposit Money Use Case Diagram:



Use Case: Withdraw Money
Perspective: Administration
Brief Description: Money is removed from a fund.
Business Goals: Money is removed from a fund for specified uses associated with the specific fund. This process may be automated.
Actors: Authorized User or automated system

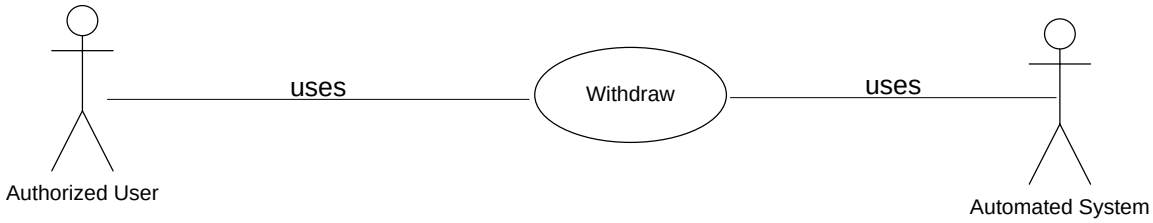
Preconditions:

1. The user must be authorized for the transaction.
2. The fund must be active.
3. The amount to withdraw must be less than the balance of the fund. Zero balance funds are not allowed.

Post-conditions:

1. The amount is reduced from the fund balance.
2. The account balance is reduced the same amount as the fund balance reduction to reflect the fund balance reduction.
3. A fund level transaction is created to record the withdrawal transaction.

Withdraw Money Use Case Diagram:



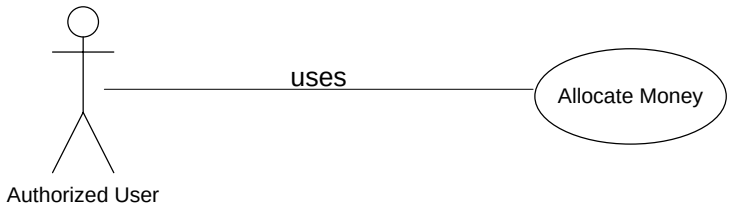
Use Case: Allocate Money
Perspective: Administration
Brief Description: New money is added to a fund; this activity automatically creates a deposit transaction.
Business Goals: All money is initially added at the account level and placed into the general fund for distribution to the other funds.
Actors: Authorized User
Preconditions:

1. The user must be authorized for the transaction.
2. An account must exist prior to adding money.
3. The amount of the deposit must be greater than zero.
4. The account must have an active status.

Post-conditions:

1. The amount of the deposit is added to the general fund.
2. The account balance is the sum of the individual account values for each fund associated with the account.
3. An account level transaction records the deposit.
4. A fund level transaction records the deposit.

Allocate Money Use Case Diagram:



Use Case: Set Goal
Perspective: A goal is associated with a fund to assist in tracking expenses and spending.
Brief Description: A goal is set for the fund. The process maintains a history of the goals set for the fund.
Business Goals: Track the goals set for a specific fund.
Actors: Authorized User
Preconditions:

1. The fund has more than zero dollar balance
2. The fund has an active status

Post-conditions:

1. A goal is associated with the fund.
2. A history of all goals set for the fund is recorded and tracked by the date of the goal.

Set Goal Use Case Diagram:

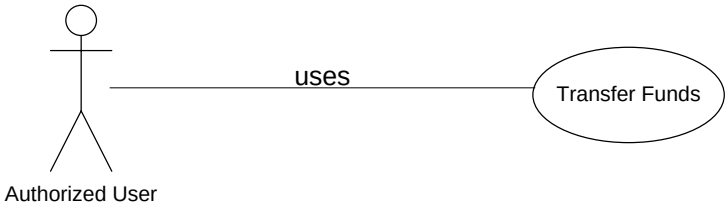
Use Case: Transfer Funds
Perspective: Administration
Brief Description: Transfer money from one fund to another fund. The source fund and the destination fund are associated with the same account.
Business Goals: Transfer money between funds
Actors: Authorized User
Preconditions:

1. The source fund is associated with the account.
2. The destination fund is associated with the account.
3. The account is open.
4. The source fund is open.
5. The destination fund is open.

Post-conditions:

1. The balance of the account has not changed.

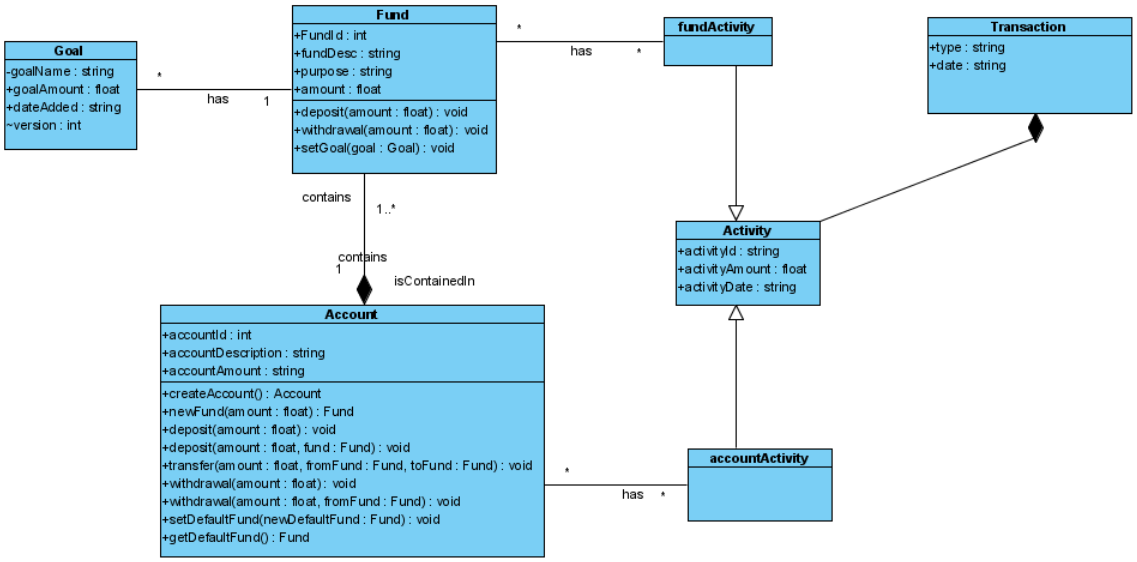
Transfer Funds Use Case Diagram:



Global Constraints

1. Each account must have at least one fund associated with it.
2. The sum of the transaction amounts associated with a fund must equal the balance of the fund.
3. The sum of the fund balances associated with the account must equal the balance of the account.
4. The sum of the account activity equals the balance of the account.

Example System UML Diagram



In this system, the bank accounts are the parent entity, and the accounts can have any number of associated funds. The sum of the balance of the funds associated with the account is the balance of the account.

Using OCL we can describe the class structure, relationships, and constraints, both global constraints and the pre- and post- conditions associated with the methods on the classes.

Assumptions:

In this model of the application, it is assumed that an account or fund is open when the account or fund balance is greater than zero.

OCL DESCRIPTION OF THE UML STRUCTURE

The code listing below provides the description of the Unified Model Language structure and the Object Constraint Language description of the constraints for each method along with the global constraints.

```
model Fund

--Classes

class Fund
attributes
  fundId       : Integer
  fundDesc     : String
  purpose      : String
  amount       : Real
operations
  deposit(amount: Real)
    pre:  amount > 0
    pre:  self.amount >= 0
    pre:  self.fundOf.accountAmount >= 0
    post: self.amount = self.amount@pre + amount
  withdrawal(amount: Real)
    pre:  amount > 0
    pre:  self.amount >= amount
    post: self.amount = self.amount@pre - amount
  setGoal(goal: Goal)
    post: self.setGoals->includes(goal)
end

class Goal
attributes
  goalName      : String
  goalAmount    : Real
  dateAdded     : String
  version       : Integer
end
```

```
class FundActivity
attributes
  ActivityId    : Integer
  activityAmount: Real
end

class Activity
attributes
  ActivityId    : Integer
  activityAmount: Real
end


class Transaction
attributes
  TransactionId : Integer
  Comments      : String
  Amount        : Real
end


class AccountActivity
attributes
  ActivityId    : Integer
  ActivityAmount: Real
end


class AccountAssociation
attributes
  AccountId    : Integer
  Activityid   : Integer
end


class Account
attributes
  accountId         : Integer
  accountDescription:String
  accountAmount     : Real
  defaultFund       : Fund
operations
  newFund(amount: Real)     : Fund
```

```
  deposit1(amount: Real, fund: Fund)
    pre: amount > 0
    pre: accountAmount >= 0
    post: accountAmount = accountAmount@pre + amount
    post: fund.amount = fund.amount@pre + amount
  deposit2(amount: Real)
    pre: amount > 0
    pre: accountAmount >= 0
    post: accountAmount = accountAmount@pre + amount
    post: defaultFund.amount = defaultFund.amount@pre +
amount
  transfer(amount: Real, fromFund: Fund, toFund: Fund)
    pre: amount > 0
    pre: self.contains->includes(fromFund)
    pre: self.contains->includes(toFund)
    pre: fromFund.amount > amount
    pre: toFund.amount >= 0
    post: accountAmount = accountAmount@pre
    post: fromFund.amount = fromFund.amount@pre - amount
    post: toFund.amount = toFund.amount@pre + amount
  withdrawal1(amount: Real)
    pre: amount > 0
    pre: accountAmount > amount
    pre: defaultFund.amount > amount
    post: accountAmount = accountAmount@pre - amount
    post: defaultFund.amount = defaultFund.amount@pre -
amount
  withdrawal2(amount: Real, fromFund: Fund)
    pre: amount > 0
    pre: accountAmount > amount
    pre: fromFund.amount > amount
    post: accountAmount = accountAmount@pre - amount
    post: fromFund.amount = fromFund.amount@pre - amount
  setDefaultFund(newDefaultFund: Fund)
    pre: self.contains->includes(newDefaultFund)
    post: defaultFund = newDefaultFund
  getDefaultFund(): Fund

end


class History
attributes
  HistoryId    : Integer
```

```
  Balance      : Real
end


-- Associations
association historyRecorded between
        History [*] role HistoryOf
        Fund[1] role detailOf
end


association containsFunds between
        Fund[1..*] role contains
        Account[1] role fundOf
end

association fundFundActivity between
        Fund[*] role hasTransactions
        FundActivity[*] role detailBy
end

association accountAccountActivity between
        Account[*] role belongsTo
        AccountActivity[*] role hasActivity
end

association transactionFundActivity between
        Transaction[1] role isContained
        FundActivity[1..*] role containsFundActivity
end

association transactionAccountActivity between
        Transaction[1] role isContained
        AccountActivity[1..*] role containsAccountActivity
end

association fundHasGoals between
        Fund[*] role forFund
        Goal[*] role setGoals
end


-- Constraints
constraints
```

```
context Fund
        --A fund number is unique
        inv distinctFund:
        Fund.allInstances->forAll(s1, s2 | s1 <> s2 implies
s1.fundId <> s2.fundId)

context Fund
        --The sum of the activity amounts associated with
an account must eqaul the fund balance
        inv fundsBalanceTransaction:
        Fund.allInstances->forAll(f1 |
f1.detailBy.activityAmount->sum() = f1.amount)

context Account
        --The account number is unique
        inv distinctAccount:
        Account.allInstances->forAll(a1, a2 |a1 <> a2
implies a1.accountId <> a2.accountId)

context Account
        --The sum of the funds equal the sum of the funds
        inv accountBalance:
        Account.allInstances->forAll(a | a.contains.amount-
>sum() = a.accountAmount)

context Account
        --The sum of the activity amounts associated with
an account must equal
        inv accountBalanceTransaction:
        Account.allInstances->forAll(a |
a.hasActivity.ActivityAmount->sum() = a.accountAmount)

context Account
        --The account must have at least one Fund
        inv atLeastOneFund:
        Account.allInstances->forAll(a | a.contains->size()
>= 1)
```

COMMENTS ABOUT THE CODE

In the code used to describe this UML model, we elected to use the field accountAmount on the Account object to indicate both the account balance and indicate with a zero or greater balance the that the account was open, rather than further complicate the code with an enumeration that would have distracted from the essential details.

Below we take a closer look at the code for two classes:

Class: Fund

The fund class has two methods:

- deposit(amount : Real).
    - Preconditions:
        - Amount parameter is greater than zero.
        - The fund is active and as such its balance is greater than or equal to zero.
        - The account associated with the fund is active and as such its balance is greater than or equal to zero.
    - Post-condition:
        - The fund balance upon the completion of the deposit is equal to the fund balance before the deposit plus the deposit amount.
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object.
        - The deposit transaction is recorded by a transaction object.

- withdrawal(amount : Real)
    - Preconditions:
        - The amount parameter is greater than zero. The system will not allow zero dollar withdrawals.
        - The fund balance is greater than the amount parameter. The system will not allow the fund balance to drop to zero, due to the stated assumption that a zero balance fund is a closed fund.
    - Post-Conditions:
        - The current balance upon the completion of the withdrawal is equal to the fund balance before the deposit minus the withdrawal amount
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object
        - The deposit transaction is recorded by a transaction object.

Class: Account

The account class is the source of most events for the system and as a result has the majority of the methods:

- newFund(amount: Real): Fund
    - This method has no preconditions and post-conditions associated with it; this method is a factory for funds.
- deposit1(amount: Real, fund: Fund)
    - Preconditions:

        - The deposit amount is greater than zero. The system will not allow a zero dollar deposit.
        - The account is active and as a result the account balance is greater than or equal to zero.
    - Post-Conditions
        - The account balance is increased by the amount of the deposit.
        - The fund balance is increased by the amount of the deposit.
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object.
        - The deposit transaction is recorded by a transaction object.
- deposit2(amount: Real)
    - This method makes use of the default fund associated with the account.
    - Preconditions
        - The deposit amount is greater than zero. The system will not allow a zero dollar deposit.
        - The account is active, and as a result the account balance is greater than or equal to zero.
    - Post-Conditions
        - The account balance is increased by the amount of the deposit.

        - The default fund balance is increased by the amount of the deposit.
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object.
        - The deposit transaction is recorded by a transaction object.
- transfer(amount: Real, fromFund: Fund, toFund: Fund)
    - Preconditions
        - The transfer amount parameter is greater than zero.
        - The fromFund is associated with the account.
        - The toFund is associated with the account.
        - The fromFund has a fund balance greater than the amount parameter.
    - Post-conditions
        - The value of the account has not changed as a result of the fund transfer.
        - The current balance of the fromFund is equal to the previous value of the fromFund minus the transfer amount.
        - The current balance of the toFund is equal to the previous value of the toFund plus the transfer amount.
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object.

        - The deposit transaction is recorded by a transaction object.
- withdrawal1(amount: Real)
    - This withdrawal method makes use of the default fund associated with the account.
    - Preconditions
        - The amount parameter is greater than zero.
        - The balance of the default fund is greater than the amount of the withdrawal.
    - Post-conditions
        - The balance of the account is equal to the previous value of the balance minus the withdrawal amount.
        - The balance of the fund is equal to the previous value of the balance minus the withdrawal amount.
        - The deposit transaction is recorded by an account activity object.
        - The deposit transaction is recorded by a fund activity object.
        - The deposit transaction is recorded by a transaction object.
- withdrawal2(amount: Real, fromFund: Fund)
    - Preconditions
        - The amount parameter is greater than zero.
        - The balance of the fromFund is greater than the amount of the withdrawal.
    - Post-conditions

- The balance of the account is equal to the previous value of the balance minus the withdrawal amount.
- The balance of the fund is equal to the previous value of the balance minus the withdrawal amount.
- The deposit transaction is recorded by an account activity object.
- The deposit transaction is recorded by a fund activity object.
- The deposit transaction is recorded by a transaction object.

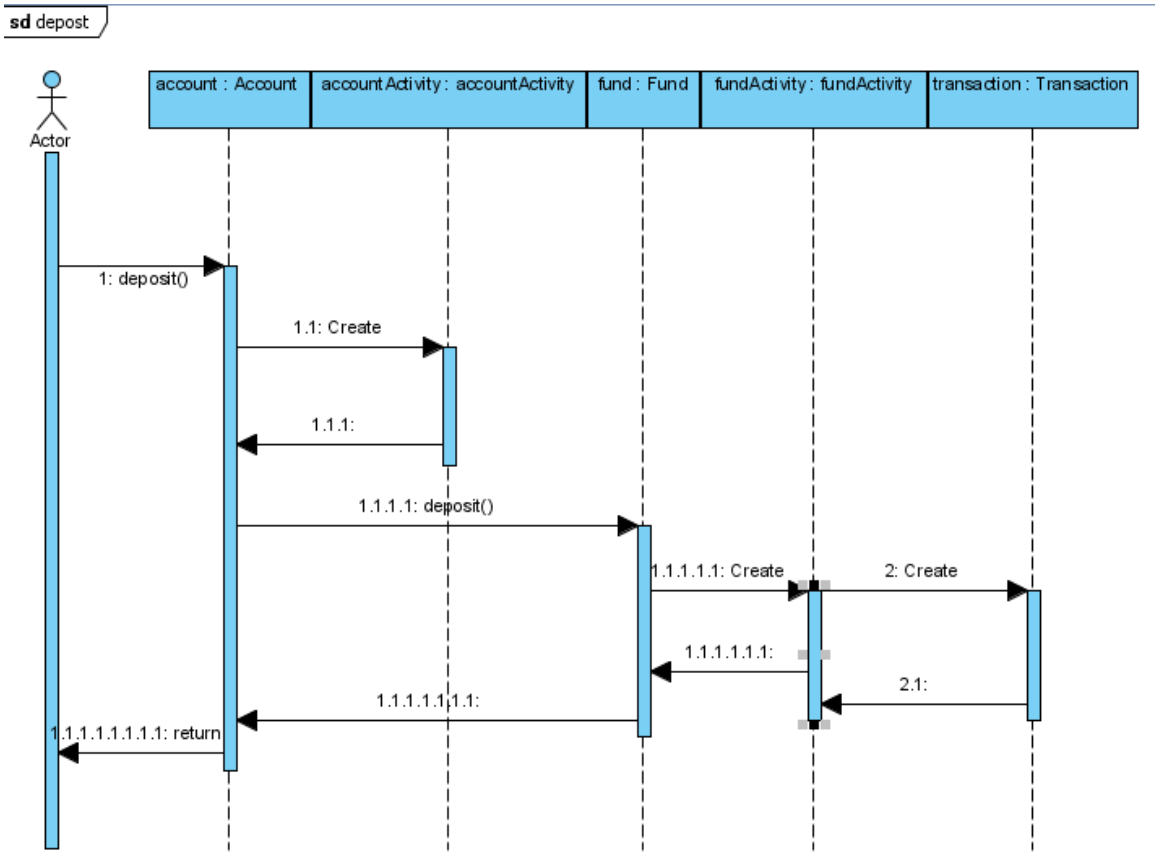
This OCL description gives the structure of the classes, the relationships, and the relationship constraints. System states, however, do not completely describe a system [7. 11]. Actions and events can be described using UML with Sequence diagrams and State Machine diagrams. Due to the nature of the application, sequence diagrams were chosen to describe the system actions. A state machine is described as the following:

> A state machine is a graph of states and transitions that describe the response of an instance of a classifier to the receipt of events. [3]

The sample presented does not have objects with long series of states and state transitions that would be displayed in a state diagram. Instead the application's objects coordinate and communicate to complete a specific task; this communication and coordination is best represented by a sequence diagram.

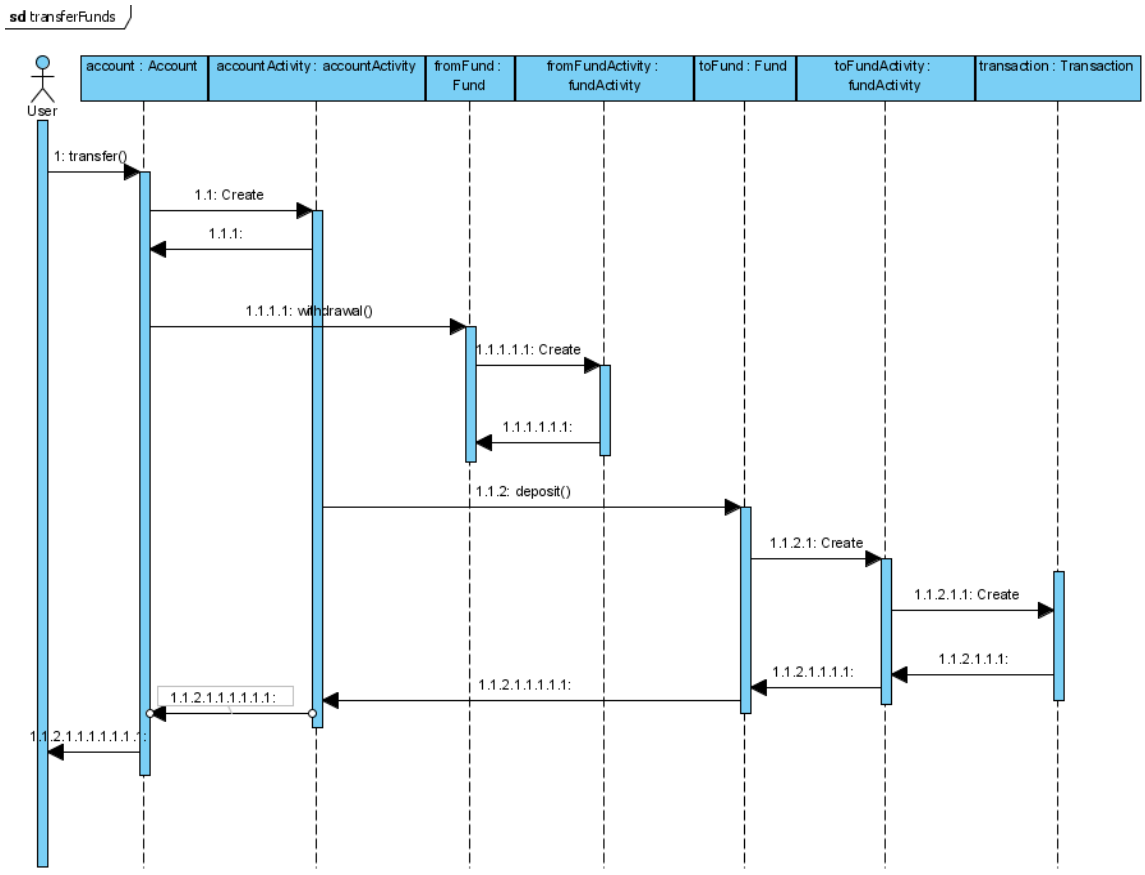
Example Application Sequence Diagram: Deposit Money

When a deposit is made it is recorded by a new instance of the accountActivity object. The deposit is placed in a fund and this fund deposit is recorded by a new instance of the fundActivity object. This fund level activity is also documented by a new instance of the transaction object that provides an audit trail between the fund and the account. These sequences of program calls are illustrated below in the deposit money sequence diagram.



Sequence Diagram: Transfer Funds

When money is transferred between funds the transfer is recorded by an update to the account instance, the transfer is recorded by an new instance of the accountActivity object. These transfers require updates to the 'fromFund', which records the update with a new instance of the fundActivity object. Similarly, the 'toFund' is updated and a new instance of the transaction object associates all of the transaction instances to provide an audit trail. These sequences of program calls are illustrated below in the transfer Funds sequence diagram.



ACTION LANGUAGES

In order to describe the life of the objects and the events associated with them, an action language is needed. The UML specification does not include an action language. However the MDA standard for Executable UML recognizes this gap and offers a few choices for a language, such Small, a language that describes explicit data flow similar to a shell script, and Tall a functional language based on action semantics [8]. The USE application used in this study makes use of a unique action language that is not part of the MDA specification or the UML specification [3, 16].

An action language encapsulates the actions detailed in sequence diagrams. The sequence diagrams for deposit and transfer are formalized into the USE action language in appendix A and appendix B.

EXAMINING THE APPLICATION

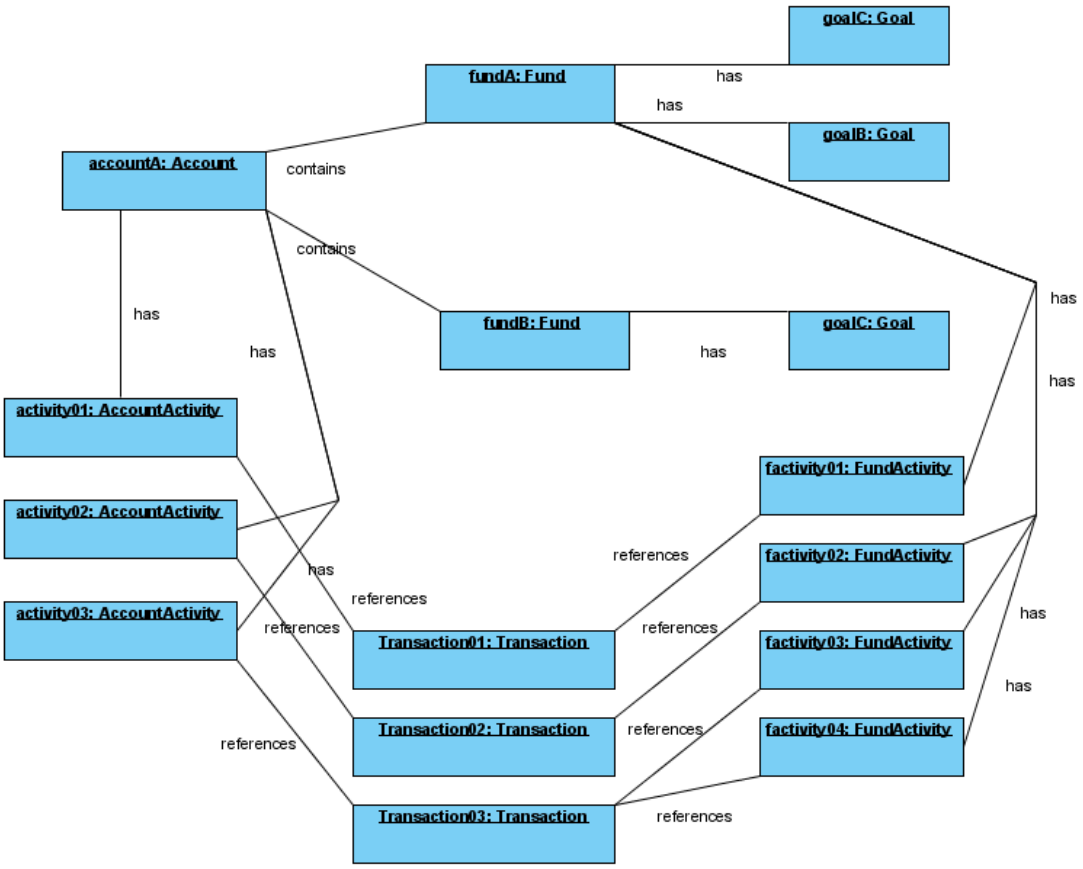
State Analysis

The application's objects, relationships, constraints and actions have been defined. To test this design and determine if the application meets the requirements, specific instances of expected states are needed for comparison and analysis.

The solution described in this example defines the proposed design for the system. During the requirements process, proposed system states were obtained. These proposed states represent instantiations of the objects from the model. These objects are represented in the requirements using object diagrams.

Example: Account deposits and fund transfers with historical records.

OBJECT DIAGRAM SYSTEM STATE 1:



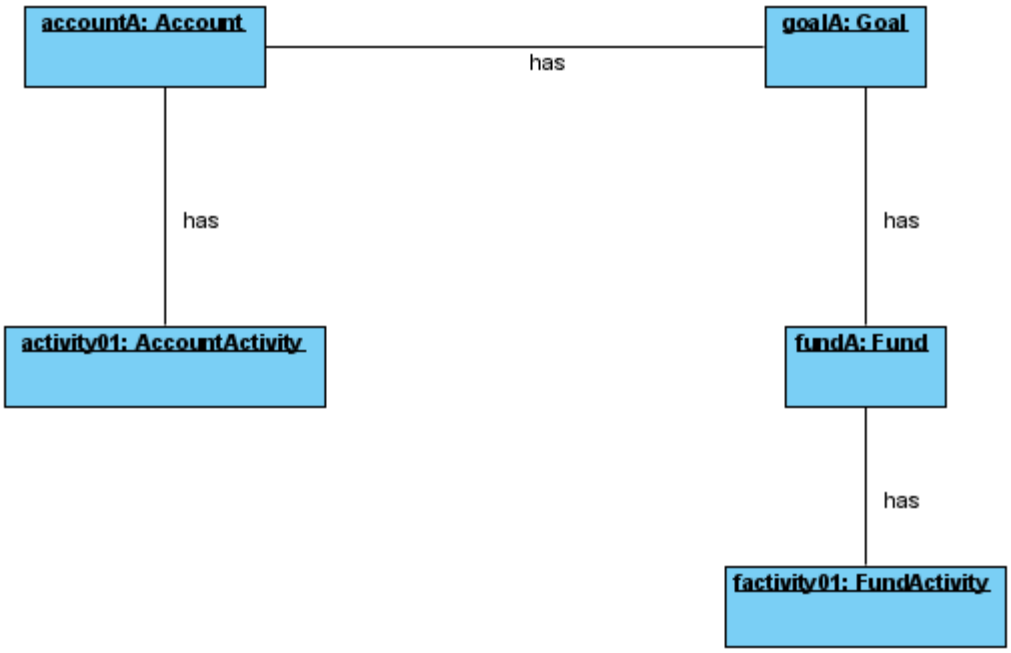
The action language code to build this object model is listed in Appendix C. This instantiation model can be examined formally to determine if given the model constraints and model definition, the instantiated objects and associations represent a valid system state.

The USE tool provides a formal evaluation of all the global constraints to determine if they are held for all object instances. The results of an error-free evaluation indicate that the system state meets all criteria given and is valid in the solution model described. The action language successfully created and maintained the relationship

definitions and function constraints; the action language code used to evaluate the model are listed in appendix D. A sample output of the validation provided by USE is listed in appendix F.

Another Proposed System State:

Object Diagram System State 2:



The action language code to build the object model shown above is listed in appendix E. This instantiation was also examined formally to determine if given the model definitions and constraints, if this state is a valid.

The formal evaluation of this state fails, indicating that the solution model does not allow for this state. The failure of this system state presents the developer with an opportunity to redefine the system model to allow the state and/or return to system stakeholders to elicit additional requirements.

Notes on USE case Action Scripts

The action script used by the USE tool is a direct scripting language, but it has significant drawbacks. The scripting language requires full knowledge of the implementation details for each class in order to make updates and simulate method calls. This level of detailed coding breaks the encapsulation of the individual classes. This can be in conflict with good software practices of encapsulation and information hiding. For example, in the cases of the methods on the Account object, the action scripting needs to manipulate the Fund objects and the associated activity objects, thereby violating object encapsulation. The action script code has global access to all objects and attributes. For this small application, this global scope works well, but as the size of an application increases and developers make use of code libraries, this could become too complex or impossible in situations where the internal implementation details of a class are not known. In these situations the libraries need independent evaluation to ensure correct functionality and correct output provided to library users..

OCL also limits the ability to document inheritance. In the original system model, the classes fundActivity and accountActivity were children of the Activity class, in which all of the fields were contained. Using the USE tool is was not possible to implement this model in OCL such that fundActivity and accountActivity inherited their fields from their parent. The USE tool would not list the attributes associated with a parent class as attributes in the child class. As a result, attempts to set values to instantiations of fundActivity or accountActivity would return errors. To get around this limitation the attributes were placed in the sub-classes used in the analysis.

The OCL parser in USE did not allow methods to have the same name but with a separate signature. In the original model of the system the method withdrawal, had two signatures, withdrawal(amount: float, fromFund: Fund) and withdrawal(amount: float),

with the latter signature assuming the use of the defaultFund attribute stored in the Account class. The OCL implementation required that these signatures be changed to withdrawal1(amount: Real) and withdrawal2(amount: Real, fromFund: Fund).

Behavior Analysis

As part of any system validation, a series of test scenarios and test scripts should be developed to exercise the application's logic. These test scripts are developed from the use cases. To determine if the solution model meets the functional requirements requested by the users, the use cases are translated into action language code. Using this action language, the use cases can be run against the model to determine if the model meets the stated functional requirements. The collection of action language scripts thus becomes a form of simulation to determine how the model reacts to the stimuli detailed in the specified use cases.

Testing the Software Model

TEST CASES

Table: Test Execution Script

Requirement	Test	Source	Test Number	Expected Results
Create Account	Create account B	Prerequisite for use cases	01	• New Account create • Default fund created
Deposit Money	Deposit $500 into account A	Use case: Deposit Money	02	• Account A balance increased to $500 • Default fund balance increased to $500
Deposit Money	Deposit $250 into account A	Use case: Deposit Money	03	• Account A balance increased to $750 • Default fund balance increased to $750
Set Goal	Set a goal of $1000 for fund A for rent on 10/15/2009	User case: Set Goal	04	• New goal created • Goal associated with the default fund
Create Fund	Create fund B	Prerequisite for user cases	05	• New fund B created • Fund B set to an initial balance of zero
Transfer Funds	Transfer $250 from fund A to fund B	Use case: Transfer funds	06	• Account balance does not change • Fund A balance is decreased by $250 to $250 • Fund B balance is increase by $250 to $250
Withdrawal	Withdrawal $50	Use case:	07	• Account balance

Funds	from fund A	Withdrawal		decreased by $50 to $700 • Fund A balance is decreased by $50 to $200
Withdrawal Funds	Withdrawal $50 from fund B	Use case Withdrawal	08	• Account balance decreased by $50 to $650 • Fund B balance is decreased by $50 to $200

These test cases are described in the action language below along with the test results.

ACTION SCRIPT FOR USE CASES

```
--Create New Account, subsequent consequence create new ---
--fund and set the new fund as the
--default fund in account class
!create accountA : Account
!set accountA.accountAmount := 0
!set accountA.accountDescription := 'General Account'
!set accountA.accountId := 1
!openter accountA newFund(0)
!create fundA : Fund
!set fundA.amount := 0
!set fundA.fundId :=1
!insert (fundA, accountA) into containsFunds
!opexit fundA
!openter accountA setDefaultFund(fundA)
!set accountA.defaultFund := fundA
!opexit

--Deposit $500 into account, subsequent consequence $500
added to fund A, activity

element
--are created
!openter accountA deposit2(500)
!openter fundA deposit(500)
!create factivity01 : FundActivity
```

```
!set factivity01.ActivityId := 1
!set factivity01.activityAmount := 500
!set fundA.amount := 500
!insert (fundA, factivity01) into fundFundActivity
!opexit
!create activity01 : AccountActivity
!set activity01.ActivityAmount := 500
!set activity01.ActivityId := 1
!insert (accountA, activity01) into accountAccountActivity
!set accountA.accountAmount := 500
!create transaction01 : Transaction
!set transaction01.TransactionId := 1
!set transaction01.Comments := 'Deposit'
!set transaction01.Amount := 500
!insert (transaction01, activity01) into
transactionAccountActivity
!insert (transaction01, factivity01) into
transactionFundActivity
!opexit

--Deposit an Additional $250 into account A
!openter accountA deposit2(250)
!openter fundA deposit(250)
!create factivity02 : FundActivity
!set factivity02.ActivityId := 2
!set factivity02.activityAmount := 250
!set fundA.amount := 750
!insert (fundA, factivity02) into fundFundActivity
!opexit
!create activity02 : AccountActivity
!set activity02.ActivityAmount := 250
!set activity02.ActivityId := 2
!insert (accountA, activity02) into accountAccountActivity
!set accountA.accountAmount := 750
!create transaction02 : Transaction
!set transaction02.TransactionId := 2
!set transaction02.Comments := 'Deposit'
!set transaction02.Amount := 250
!insert (transaction02, activity02) into
transactionAccountActivity
!insert (transaction02, factivity02) into
transactionFundActivity
!opexit
```

```
--Set goal associated with fundA
!create goalA : Goal
!set goalA.goalName :='Rent'
!set goalA.goalAmount := 1000
!set goalA.dateAdded := '20091015'
!set goalA.version := 1
!openter fundA setGoal(goalA)
!insert (fundA, goalA) into fundHasGoals
!opexit

--create new fund fundB
!openter accountA newFund(0)
!create fundB : Fund
!set fundB.amount := 0
!set fundB.fundId := 2
!insert (fundB, accountA) into containsFunds
!opexit fundB

--Transfer $250 from fundA to fundB, create subsequent
activity
!openter accountA transfer(250, fundA, fundB)
!openter fundA withdrawal(250)
!set fundA.amount := 250
!create factivity03 :FundActivity
!set factivity03.ActivityId := 3
!set factivity03.activityAmount := -250
!insert (fundA, factivity03) into fundFundActivity
!set fundA.amount := 500
!opexit
!openter fundB deposit(250)
!create factivity04 : FundActivity
!set factivity04.ActivityId := 4
!set factivity04.activityAmount := 250
!insert (fundB, factivity04) into fundFundActivity
!set fundB.amount := 250
!opexit
!create activity03 : AccountActivity
!set activity03.ActivityAmount := 0
!set activity03.ActivityId := 3
!insert (accountA, activity03) into accountAccountActivity
!create transaction03 : Transaction
!set transaction03.TransactionId := 3
!set transaction03.Comments := 'Transfer'
!set transaction03.Amount := 250
```

```
!insert (transaction03, activity03) into
transactionAccountActivity
!insert (transaction03, factivity03) into
transactionFundActivity
!insert (transaction03, factivity04) into
transactionFundActivity
!opexit

--Get a reference to the default fund
!openter accountA getDefaultFund()
!opexit accountA.defaultFund

--withdrawal $50 from the default fund
!openter accountA withdrawal1(50)
!openter fundA withdrawal(50)
!create factivity05 : FundActivity
!set factivity05.ActivityId := 7
!set factivity05.activityAmount := -50
!insert (fundA, factivity05) into fundFundActivity
!set fundA.amount := 450
!opexit
!set accountA.accountAmount := 700
!create activity04 : AccountActivity
!set activity04.ActivityAmount := -50
!set activity04.ActivityId := 4
!insert (accountA, activity04) into accountAccountActivity
!create transaction04 : Transaction
!set transaction04.TransactionId := 4
!set transaction04.Comments := 'Withdrawal'
!set transaction04.Amount := -50
!insert(transaction04, activity04) into
transactionAccountActivity
!insert(transaction04, factivity05) into
transactionFundActivity
!opexit

--Withdrawal $50 from fund B, note that this fund is not
the default fund.
!openter accountA withdrawal2(50, fundB)
!openter fundB withdrawal(50)
!create factivity06 :FundActivity
!set factivity06.ActivityId := 6
!set factivity06.activityAmount := -50
!insert (fundB, factivity06) into fundFundActivity
```

```
!set fundB.amount := 200
!opexit
!set accountA.accountAmount := 650
!create activity05 : AccountActivity
!set activity05.ActivityAmount := -50
!set activity05.ActivityId := 5
!insert (accountA, activity05) into accountAccountActivity
!create transaction05 : Transaction
!set transaction05.TransactionId := 5
!set transaction05.Comments := 'Withdrawal'
!set transaction05.Amount := -50
!insert(transaction05, activity05) into
transactionAccountActivity
!insert(transaction05, factivity06) into
transactionFundActivity
!opexit
```

Action Script Description

The second section of the action script provides detail of the sequence of interaction for a deposit into a fund. The call to the function deposit2 is simulated by the command,

!openter accountA deposit2(500)

Similarly the call to the function deposit on the fund object is simulated. Inside this function a new instance of the FundActivity, AccountActivity and Transaction objects are created and the properties for these new instances are set. In addition, the associations to other objects are created using the associations defined in the OCL description of the structure. The correct use of these association will be confirmed using the defined constraints for these associations.

Conclusions

Using the USE specification tool it is possible to sufficiently describe the states and behavior of the example application, with UML as a standardized language. However, the limits of the formal model notation required breaking the encapsulation of the objects in the model. Specifically, to describe the guard conditions for the classes, access fields in other classes. The creation of universal constraints also required global access to all of the instance attributes for all of the classes. It is ironic that in order to add rigor and formal descriptions to the model, the authors were required to break the encapsulation and abstractions that were built.

The example was small in scope and the number of classes was easily manageable; for such a system it is possible to know the global state of the system. As systems increase in size and complexity, it will not always be feasible to know the global state of the system. A large example would likely incorporate system libraries or code from other packages, for which the modelers would have little or no knowledge of their inner workings to develop OCL conditions and Action Scripts. This is further complicated when a number of packages or libraries are involved, since it may be necessary to examine internals to understand package and library interaction to build a picture of global system state.

Using this process of a more formal description of the system provided valuable feedback on possible system states inferred from the design and its structure, but this approach provided limited value for understanding overall system behavior. The techniques used to examine system behavior scripted the actions, and it was possible to set a correct outcome for a specific test. A trace simulates the stimulus associated with the users changing the state of system objects and/or the values of object fields. Rather

than define functions that performed the addition and subtraction to modify the state, separate action scripts were executed, in which the object's state was modified and the correct balances were set. From a unit testing point of view, these results have limited value when the scripting language sets the output value to the value of the expected results.

In these trace executions the tester can script how he believes the objects will behave. The model definition provided by the UML with OCL extensions is considered valid if the behavior does not violate the system constraints. By running individual traces, the validation results can help the modeler pinpoint failed conditions. These trace executions provide feedback regarding the overall design by testing the model's response to the stimuli that will be provided from the users.

A drawback of this process is the skill level required to create the action scripts associated with test cases. The resource required to develop the test cases would need intimate knowledge of the system design details; however, this task is often assigned to staff members with little technical background [11].

These tests performed on the system model provided valuable feedback to the system designer at an early point in the development process. Unfortunately, it is believed that the limitations of the global view of the system state, the inability to make use of other packages or libraries, and the increased skill level of the staff members assigned to develop model test scripts will limit the use of this technique to validating system response to specific stimuli.

The greatest promise for this technology lies in its integration into test harness frameworks. The UML description of the system along with the OCL constraints could be used to generate a significant number of tests for automated unit testing tools. These unit tests could be run automatically during incremental builds of the system and provide

feedback on the code used to implement the model, ensuring that the code fits the original system design. The nature of unit testing is such that tests are completed one module or one class at a time, thereby reducing the impact of some of the process limitations.

The use of models improves the software development process by creating a higher level of abstraction. Specifically, UML provides valuable assistance in documenting software architectures and allows for greater abstraction and encapsulation of detail. But the modeling and validation approach investigated in this report is not enough to rigorously test the state or behavior of a system without breaking down some of the encapsulation built to hide details and increase the level of abstraction. Instead these tools provide a means to model system state and test the software model's response to stimulus. This methodology is also limited by the maturity of the current tools and processes. These processes can make a more immediate impact on unit testing procedures to ensure the code developed to implement the model maintains the properties of the original design.

Appendix A – Action Script for Deposit

```
!openter accountA deposit2(500)
!openter fundA deposit(500)
!create factivity01 : FundActivity
!set factivity01.ActivityId := 1
!set factivity01.activityAmount := 500
!set fundA.amount := 500
!insert (fundA, factivity01) into fundFundActivity
!opexit
!create activity01 : AccountActivity
!set activity01.ActivityAmount := 500
!set activity01.ActivityId := 1
!insert (accountA, activity01) into accountAccountActivity
!set accountA.accountAmount := 500
!create transaction01 : Transaction
!set transaction01.TransactionId := 1
!set transaction01.Comments := 'Deposit'
!set transaction01.Amount := 500
!insert (transaction01, activity01) into
transactionAccountActivity
!insert (transaction01, factivity01) into
transactionFundActivity
!opexit
```

Appendix B – Action Script for Transfer

```
!openter accountA transfer(250, fundA, fundB)
!openter fundA withdrawal(250)
!set fundA.amount := 250
!create factivity03 :FundActivity
!set factivity03.ActivityId := 3
!set factivity03.activityAmount := -250
!insert (fundA, factivity03) into fundFundActivity
!set fundA.amount := 500
!opexit
!openter fundB deposit(250)
!create factivity04 : FundActivity
!set factivity04.ActivityId := 4
!set factivity04.activityAmount := 250
!insert (fundB, factivity04) into fundFundActivity
!set fundB.amount := 250
!opexit
```

```
!create activity03 : AccountActivity
!set activity03.ActivityAmount := 0
!set activity03.ActivityId := 3
!insert (accountA, activity03) into accountAccountActivity
!create transaction03 : Transaction
!set transaction03.TransactionId := 3
!set transaction03.Comments := 'Transfer'
!set transaction03.Amount := 250
!insert (transaction03, activity03) into
transactionAccountActivity
!insert (transaction03, factivity03) into
transactionFundActivity
!insert (transaction03, factivity04) into
transactionFundActivity
!opexit
```

Appendix C – Action Script to Create System State 1

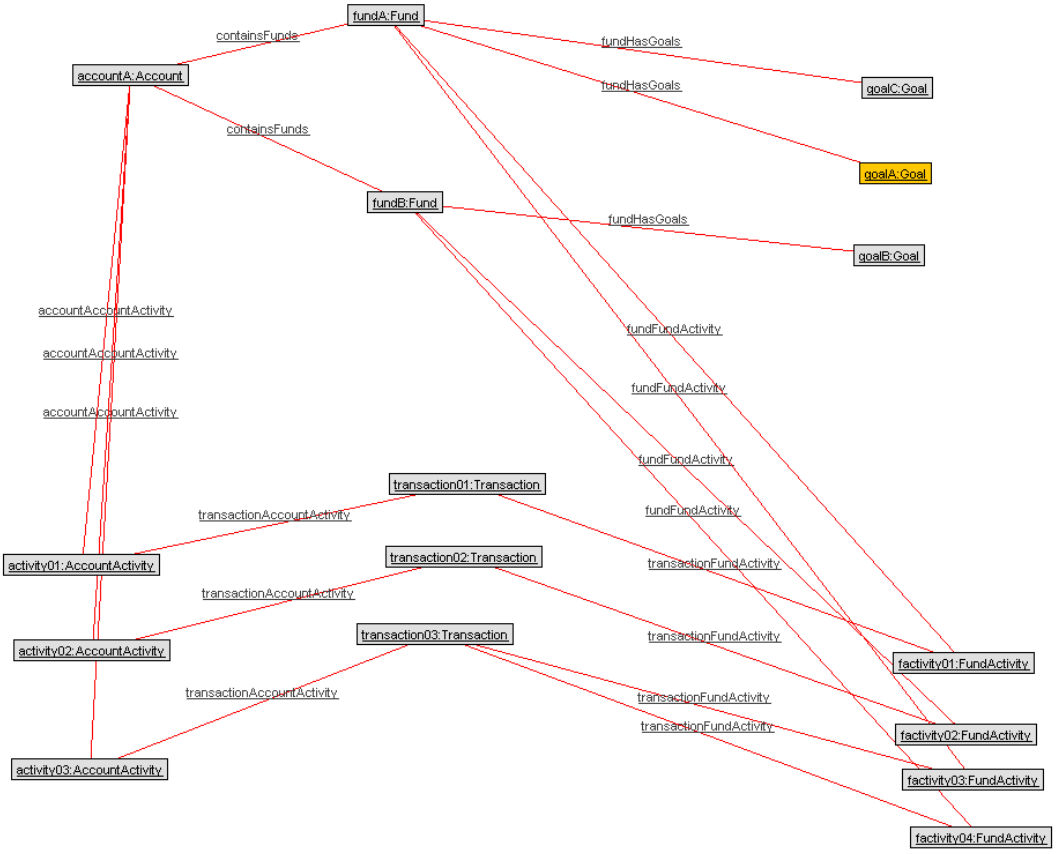
```
--Create New Account, subsequent consequence create new
fund and set the new

fund as the
--default fund in account class
!create accountA : Account
!set accountA.accountAmount := 0
!set accountA.accountDescription := 'General Account'
!set accountA.accountId := 1
!openter accountA newFund(0)
!create fundA : Fund
!set fundA.amount := 0
!set fundA.fundId := 1
!insert (fundA, accountA) into containsFunds
!opexit fundA
!openter accountA setDefaultFund(fundA)
!set accountA.defaultFund := fundA
!opexit

--Deposit $500 into account, subsequent consequence $500
added to fund A,

activity element
--are created
!openter accountA deposit2(500)
!openter fundA deposit(500)
```

```
!create factivity01 : FundActivity
!set factivity01.ActivityId := 1
!set factivity01.activityAmount := 500
!set fundA.amount := 500
!insert (fundA, factivity01) into fundFundActivity
!opexit
!create activity01 : AccountActivity
!set activity01.ActivityAmount := 500
!set activity01.ActivityId := 1
!insert (accountA, activity01) into accountAccountActivity
!set accountA.accountAmount := 500
!create transaction01 : Transaction
!set transaction01.TransactionId := 1
!set transaction01.Comments := 'Deposit'
!set transaction01.Amount := 500
!insert (transaction01, activity01) into
transactionAccountActivity
!insert (transaction01, factivity01) into
transactionFundActivity
!opexit

--Create fundB
!openter accountA newFund(0)
!create fundB : Fund
!set fundB.amount := 0
!set fundB.fundId := 2
!insert (fundB, accountA) into containsFunds
!opexit fundB

--Deposit an Additional $250 into fundB
!openter accountA deposit1(250, fundB)
!openter fundB deposit(250)
!create factivity02 : FundActivity
!set factivity02.ActivityId := 2
!set factivity02.activityAmount := 250
!set fundB.amount := 250
!insert (fundB, factivity02) into fundFundActivity
!opexit
!create activity02 : AccountActivity
!set activity02.activityAmount := 250
!set activity02.ActivityId := 2
!insert (accountA, activity02) into accountAccountActivity
!set accountA.accountAmount := 750
!create transaction02 : Transaction
```

```
!set transaction02.TransactionId := 2
!set transaction02.Comments := 'Deposit'
!set transaction02.Amount := 250
!insert (transaction02, activity02) into
transactionAccountActivity
!insert (transaction02, factivity02) into
transactionFundActivity
!opexit

--Set goal associated with fundA
!create goalA : Goal
!set goalA.goalName :='Rent'
!set goalA.goalAmount := 1000
!set goalA.dateAdded := '20091015'
!set goalA.version := 1
!openter fundA setGoal(goalA)
!insert (fundA, goalA) into fundHasGoals
!opexit

--Set goal associated with fundB
!create goalB : Goal
!set goalB.goalName :='Salaries'
!set goalB.goalAmount := 2000
!set goalB.dateAdded := '20091020'
!set goalB.version := 1
!openter fundB setGoal(goalB)
!insert (fundB, goalB) into fundHasGoals
!opexit

--Transfer $250 from fundA to fundB, create subsequent
activity
!openter accountA transfer(250, fundA, fundB)
!openter fundA withdrawal(250)
!set fundA.amount := 250
!create factivity03 :FundActivity
!set factivity03.ActivityId := 3
!set factivity03.activityAmount := -250
!insert (fundA, factivity03) into fundFundActivity
!opexit
!openter fundB deposit(250)
!create factivity04 : FundActivity
!set factivity04.ActivityId := 4
!set factivity04.activityAmount := 250
!insert (fundB, factivity04) into fundFundActivity
```

```
!set fundB.amount := 500
!opexit
!create activity03 : AccountActivity
!set activity03.ActivityAmount := 0
!set activity03.ActivityId := 3
!insert (accountA, activity03) into accountAccountActivity
!create transaction03 : Transaction
!set transaction03.TransactionId := 3
!set transaction03.Comments := 'Transfer'
!set transaction03.Amount := 250
!insert (transaction03, activity03) into
transactionAccountActivity
!insert (transaction03, factivity03) into
transactionFundActivity
!insert (transaction03, factivity04) into
transactionFundActivity
!opexit

--update goal associated with fundA
!create goalC : Goal
!set goalB.goalName :='Rent'
!set goalB.goalAmount := 1200
!set goalB.dateAdded := '20091020'
!set goalB.version := 2
!openter fundA setGoal(goalC)
!insert (fundA, goalC) into fundHasGoals
!opexit
```

Object Diagram

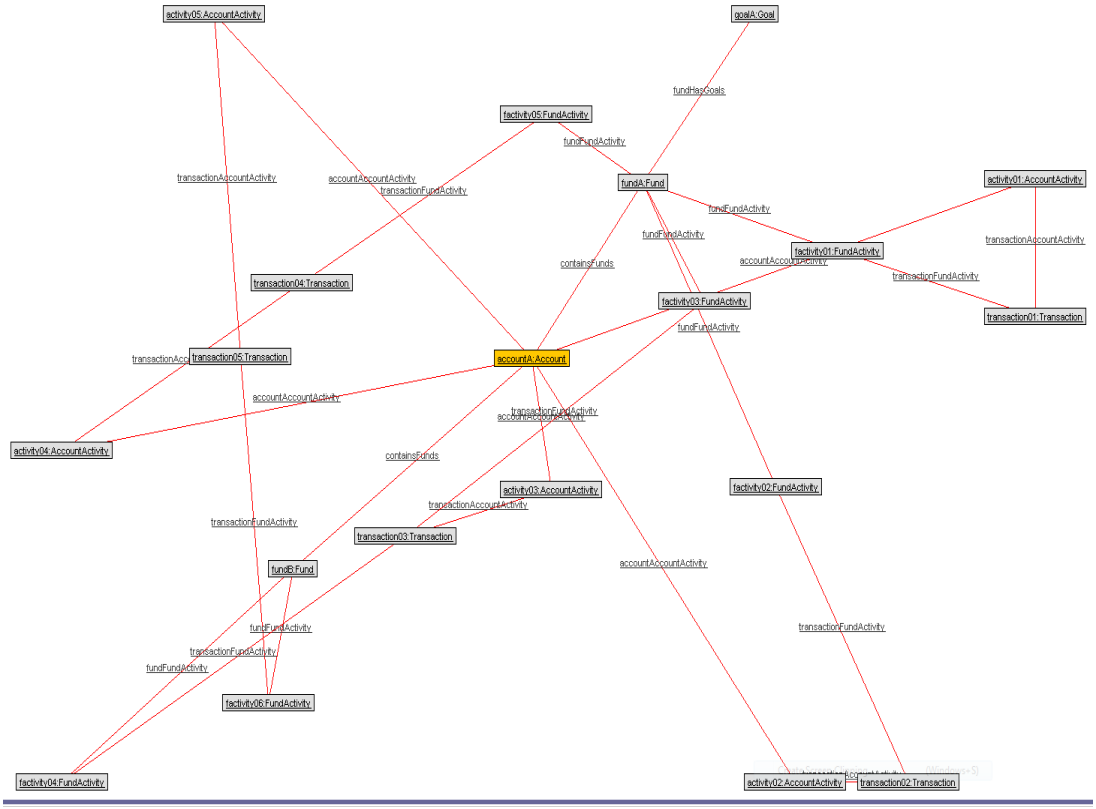


Appendix D – Action Script to Create System State 2

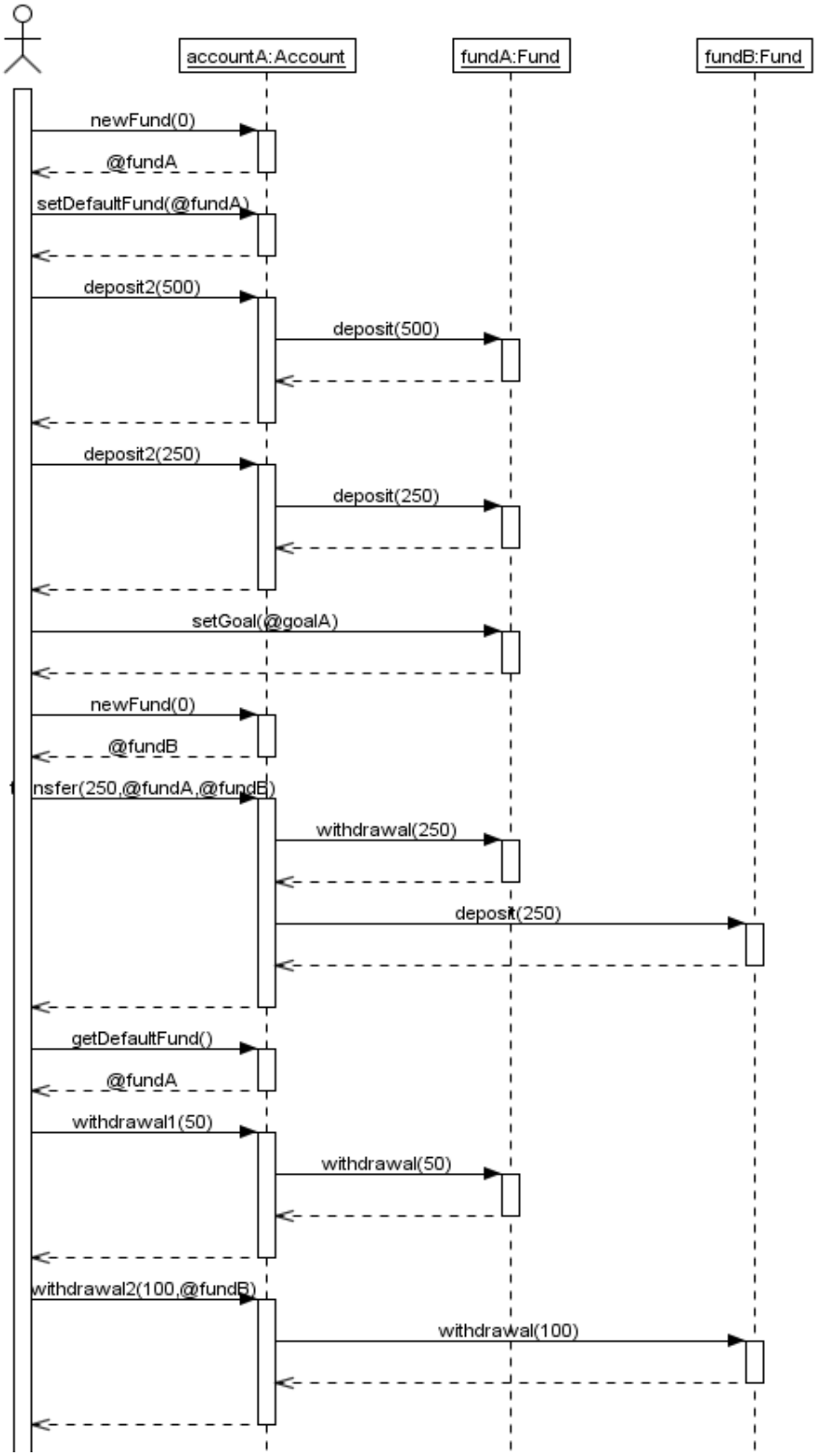
```
--create account Object
!create accountA : Account
!set accountA.accountAmount := 0
!set accountA.accountDescription := 'General Account'
!set accoundA.accountId := 1

--create other objects
!create goal1 : Goal
!create fundA : Fund
!create factivity01 : FundActivity
!activity01 : ActivityActivity

--create associatiations
!insert (accountA, goal1) into containsFunds
```

```
!insert (fundA, goal1) into containsFunds
!insert (accountA, activity) into fundFundActivity
```

Appendix E - Object Diagram from Use after Execution of Test Case



SEQUENCE DIAGRAM FROM USE AFTER EXECUTION



Appendix F – Unit Test Script Validation Ouput from USE Tool

```
use version 2.1.0, Copyright (C) 1999-2004 Mark Richters
use> read FundOCL.cmd
FundOCL.cmd> --Create New Account, subsequent consequence create new
                        fund and se
t the new fund as the
FundOCL.cmd> --default fund in account class
FundOCL.cmd> !create accountA : Account
FundOCL.cmd> !set accountA.accountAmount := 0
FundOCL.cmd> !set accountA.accountDescription := 'General Account'
FundOCL.cmd> !set accountA.accountId := 1
FundOCL.cmd> !openter accountA newFund(0)
FundOCL.cmd> !create fundA : Fund
FundOCL.cmd> !set fundA.amount := 0
FundOCL.cmd> !set fundA.fundId := 1
FundOCL.cmd> !insert (fundA, accountA) into containsFunds
FundOCL.cmd> !opexit fundA
FundOCL.cmd> !openter accountA setDefaultFund(fundA)
precondition `pre20' is true
FundOCL.cmd> !set accountA.defaultFund := fundA
FundOCL.cmd> !opexit
postcondition `post15' is true
FundOCL.cmd>
FundOCL.cmd> --Deposit $500 into account, subsequent consequence $500
                        added to f
und A, activity element
FundOCL.cmd> --are created
FundOCL.cmd> !openter accountA deposit2(500)
precondition `pre8' is true
precondition `pre9' is true
FundOCL.cmd> !openter fundA deposit(500)
precondition `pre1' is true
precondition `pre2' is true
precondition `pre3' is true
FundOCL.cmd> !create factivity01 : FundActivity
FundOCL.cmd> !set factivity01.ActivityId := 1
FundOCL.cmd> !set factivity01.activityAmount := 500
FundOCL.cmd> !set fundA.amount := 500
FundOCL.cmd> !insert (fundA, factivity01) into fundFundActivity
FundOCL.cmd> !opexit
postcondition `post1' is true
FundOCL.cmd> !create activity01 : AccountActivity
FundOCL.cmd> !set activity01.ActivityAmount := 500
```

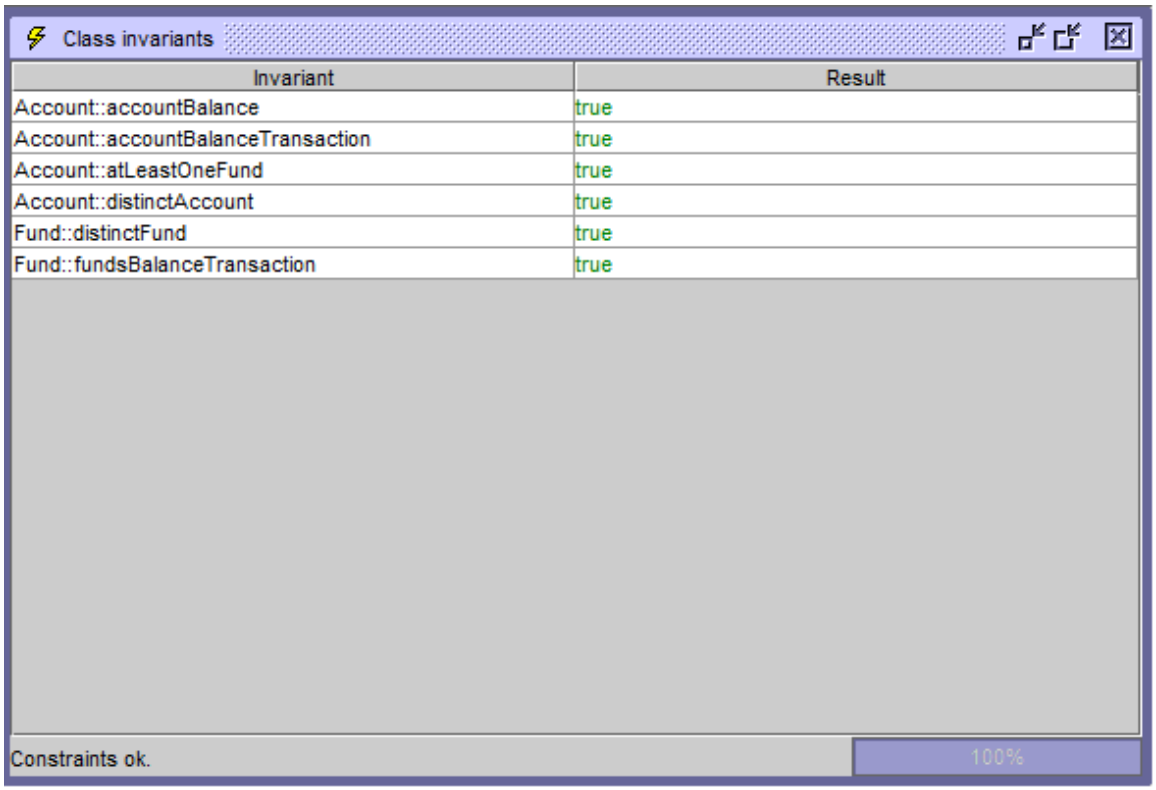
```
FundOCL.cmd> !set activity01.ActivityId := 1
FundOCL.cmd> !insert (accountA, activity01) into accountAccountActivity
FundOCL.cmd> !set accountA.accountAmount := 500
FundOCL.cmd> !create transaction01 : Transaction
FundOCL.cmd> !set transaction01.TransactionId := 1
FundOCL.cmd> !set transaction01.Comments := 'Deposit'
FundOCL.cmd> !set transaction01.Amount := 500
FundOCL.cmd> !insert (transaction01, activity01) into transactionAccountActivity

FundOCL.cmd> !insert (transaction01, factivity01) into transactionFundActivity
FundOCL.cmd> !opexit
postcondition `post6' is true
postcondition `post7' is true
FundOCL.cmd>
FundOCL.cmd> --Deposit an Additional $250 into account A
FundOCL.cmd> !openter accountA deposit2(250)
precondition `pre8' is true
precondition `pre9' is true
FundOCL.cmd> !openter fundA deposit(250)
precondition `pre1' is true
precondition `pre2' is true
precondition `pre3' is true
FundOCL.cmd> !create factivity02 : FundActivity
FundOCL.cmd> !set factivity02.ActivityId := 2
FundOCL.cmd> !set factivity02.activityAmount := 250
FundOCL.cmd> !set fundA.amount := 750
FundOCL.cmd> !insert (fundA, factivity02) into fundFundActivity
FundOCL.cmd> !opexit
postcondition `post1' is true
FundOCL.cmd> !create activity02 : AccountActivity
FundOCL.cmd> !set activity02.ActivityAmount := 250
FundOCL.cmd> !set activity02.ActivityId := 2
FundOCL.cmd> !insert (accountA, activity02) into accountAccountActivity
FundOCL.cmd> !set accountA.accountAmount := 750
FundOCL.cmd> !create transaction02 : Transaction
FundOCL.cmd> !set transaction02.TransactionId := 2
FundOCL.cmd> !set transaction02.Comments := 'Deposit'
FundOCL.cmd> !set transaction02.Amount := 250
FundOCL.cmd> !insert (transaction02, activity02) into transactionAccountActivity

FundOCL.cmd> !insert (transaction02, factivity02) into transactionFundActivity
FundOCL.cmd> !opexit
postcondition `post6' is true
postcondition `post7' is true
```

```
FundOCL.cmd>
FundOCL.cmd> --Set goal associated with fundA
FundOCL.cmd> !create goalA : Goal
FundOCL.cmd> !set goalA.goalName :='Rent'
FundOCL.cmd> !set goalA.goalAmount := 1000
FundOCL.cmd> !set goalA.dateAdded := '20091015'
FundOCL.cmd> !set goalA.version := 1
FundOCL.cmd> !openter fundA setGoal(goalA)
FundOCL.cmd> !insert (fundA, goalA) into fundHasGoals
FundOCL.cmd> !opexit
postcondition `post3' is true
FundOCL.cmd>
FundOCL.cmd> --create new fund fundB
FundOCL.cmd> !openter accountA newFund(0)
FundOCL.cmd> !create fundB : Fund
FundOCL.cmd> !set fundB.amount := 0
FundOCL.cmd> !set fundB.fundId = 2
<input>:1:20: expecting ':=', found 'end of file or input'
FundOCL.cmd> !insert (fundB, accountA) into containsFunds
FundOCL.cmd> !opexit fundB
FundOCL.cmd>
FundOCL.cmd> --Transfer $250 from fundA to fundB, create subsequent activity
FundOCL.cmd> !openter accountA transfer(250, fundA, fundB)
precondition `pre10' is true
precondition `pre11' is true
precondition `pre12' is true
precondition `pre13' is true
precondition `pre14' is true
FundOCL.cmd> !openter fundA withdrawal(250)
precondition `pre4' is true
precondition `pre5' is true
FundOCL.cmd> !set fundA.amount := 250
FundOCL.cmd> !create factivity03 :FundActivity
FundOCL.cmd> !set factivity03.ActivityId := 3
FundOCL.cmd> !set factivity03.activityAmount := -250
FundOCL.cmd> !insert (fundA, factivity03) into fundFundActivity
FundOCL.cmd> !set fundA.amount := 500
FundOCL.cmd> !opexit
postcondition `post2' is true
FundOCL.cmd> !openter fundB deposit(250)
precondition `pre1' is true
precondition `pre2' is true
precondition `pre3' is true
FundOCL.cmd> !create factivity04 : FundActivity
```

```
FundOCL.cmd> !set factivity04.ActivityId := 4
FundOCL.cmd> !set factivity04.activityAmount := 250
FundOCL.cmd> !insert (fundB, factivity04) into fundFundActivity
FundOCL.cmd> !set fundB.amount := 250
FundOCL.cmd> !opexit
postcondition `post1' is true
FundOCL.cmd> !create activity03 : AccountActivity
FundOCL.cmd> !set activity03.ActivityAmount := 0
FundOCL.cmd> !set activity03.ActivityId := 3
FundOCL.cmd> !insert (accountA, activity03) into accountAccountActivity
FundOCL.cmd> !create transaction03 : Transaction
FundOCL.cmd> !set transaction03.TransactionId := 3
FundOCL.cmd> !set transaction03.Comments := 'Transfer'
FundOCL.cmd> !set transaction03.Amount := 250
FundOCL.cmd> !insert (transaction03, activity03) into transactionAccountActivity

FundOCL.cmd> !insert (transaction03, factivity03) into transactionFundActivity
FundOCL.cmd> !insert (transaction03, factivity04) into transactionFundActivity
FundOCL.cmd> !opexit
postcondition `post8' is true
postcondition `post9' is true
postcondition `post10' is true
FundOCL.cmd>
FundOCL.cmd> --Get a reference to the default fund
FundOCL.cmd> !openter accountA getDefaultFund()
FundOCL.cmd> !opexit accountA.defaultFund
FundOCL.cmd>
FundOCL.cmd> --withdrawal $50 from the default fund
FundOCL.cmd> !openter accountA withdrawal1(50)
precondition `pre15' is true
precondition `pre16' is true
FundOCL.cmd> !openter fundA withdrawal(50)
precondition `pre4' is true
precondition `pre5' is true
FundOCL.cmd> !create factivity05 : FundActivity
FundOCL.cmd> !set factivity05.ActivityId := 7
FundOCL.cmd> !set factivity05.activityAmount := -50
FundOCL.cmd> !insert (fundA, factivity05) into fundFundActivity
FundOCL.cmd> !set fundA.amount := 450
FundOCL.cmd> !opexit
postcondition `post2' is true
FundOCL.cmd> !set accountA.accountAmount := 700
FundOCL.cmd> !create activity04 : AccountActivity
FundOCL.cmd> !set activity04.ActivityAmount := -50
```

```
FundOCL.cmd> !set activity04.ActivityId := 4
FundOCL.cmd> !insert (accountA, activity04) into accountAccountActivity
FundOCL.cmd> !create transaction04 : Transaction
FundOCL.cmd> !set transaction04.TransactionId := 4
FundOCL.cmd> !set transaction04.Comments := 'Withdrawal'
FundOCL.cmd> !set transaction04.Amount := -50
FundOCL.cmd> !insert(transaction04, activity04) into transactionAccountActivity
FundOCL.cmd> !insert(transaction04, factivity05) into transactionFundActivity
FundOCL.cmd> !opexit
postcondition `post11' is true
postcondition `post12' is true
FundOCL.cmd>
FundOCL.cmd> --Withdrawal $50 from fund B, note that this fund is not the
                        defaul
t fund.
FundOCL.cmd> !openter accountA withdrawal2(50, fundB)
precondition `pre17' is true
precondition `pre18' is true
precondition `pre19' is true
FundOCL.cmd> !openter fundB withdrawal(50)
precondition `pre4' is true
precondition `pre5' is true
FundOCL.cmd> !create factivity06 :FundActivity
FundOCL.cmd> !set factivity06.ActivityId := 6
FundOCL.cmd> !set factivity06.activityAmount := -50
FundOCL.cmd> !insert (fundB, factivity06) into fundFundActivity
FundOCL.cmd> !set fundB.amount := 200
FundOCL.cmd> !opexit
postcondition `post2' is true
FundOCL.cmd> !set accountA.accountAmount := 650
FundOCL.cmd> !create activity05 : AccountActivity
FundOCL.cmd> !set activity05.ActivityAmount := -50
FundOCL.cmd> !set activity05.ActivityId := 5
FundOCL.cmd> !insert (accountA, activity05) into accountAccountActivity
FundOCL.cmd> !create transaction05 : Transaction
FundOCL.cmd> !set transaction05.TransactionId := 5
FundOCL.cmd> !set transaction05.Comments := 'Withdrawal'
FundOCL.cmd> !set transaction05.Amount := -50
FundOCL.cmd> !insert(transaction05, activity05) into transactionAccountActivity
FundOCL.cmd> !insert(transaction05, factivity06) into transactionFundActivity
FundOCL.cmd> !opexit
postcondition `post13' is true
postcondition `post14' is true
FundOCL.cmd>
```

use>


Class invariants
Invariant
Result
Account::accountBalance
true
Account::accountBalanceTransaction
true
Account::atLeastOneFund
true
Account::distinctAccount
true
Fund::distinctFund
true
Fund::fundsBalanceTransaction
true
Constraints ok.
100%

References

1. Taylor, Richard N., et al. 2010. Software Architecture, Foundations, Theory, and Practice, John Wiley and Sons Press

2. Fowler, Martin, 2004. UML Distilled, A Brief Guide to the Standard Object Modeling Language, 3rd ed. Addison-Wesley

3. Rumbaugh, James, Booch G., Jacobsen I., 2005. The Unified Modeling Language Reference Manual, 2nd ed. Addison-Wesley

4. Clark, Tony, et al. 2001. Object Modeling with the OCL, The Rationale behind the Object Constraint Language, Lecture Notes in Computer Science, Springer

5. Paulk, Mark, et al. 1995. The Capability Maturity Model, Guidelines for Improving the Software Process, Addison Wesley

6. Podeswa, Howard, 2008. UML for the IT Business Analyst, Course Technology

7. Clements, Paul, 2008. Documenting Software Architectures, Views and Beyond, Addison-Wesley

8. Kleppe, Anneke, Warmer, Jos, Bast, Wim., 2003. MDA Explained, The Model Driven Architecture: Practice and Promise, Addison-Wesley

9. Mellor, Stephen J., Balcer Marc J., 2002. Executable UML, A Foundation for Model-Driven Architecture, Addison-Wesley

10. Warmer, Jos, Kleppe, Anne., 2003. The Object Constraint Language, 2nd ed. Addison-Wesley

11. Barber, K. Suzanne., 2009, Lecture Notes

12. Barber, K Suzanne, Graser, Tom, Holt, Jim., 2003. Evaluating Dynamic Correctness Properties of Domain Reference Architectures, The Journal of Systems and Software

13. Fowler, Martin., 2005. Language Workbenches and Model Driven Architecture, MartinFowler.com, http://martinfowler.com/articles/mdaLanguageWorkbench.html

14. Brown, Alan., Conallen, Jim., 2005. An Introduction to Model-Driven Architecture, IBM Press, http://www.ibm.com/developerworks/rational/library/apr05/brown/index.html

15. Frankel, David S., 2006, A Response to Forrester, MDA Journal, April 2006

16. Database Systems Group, Bremen University, 2007, USE – A UML based Specification Environment.

17. Gogolla, Martin, Buttner, Fabian, Richters, Mark, 2007, USE: A UML-Based Specification Environment for Validating UML and OCL, Elsevier Science, June 20, 2007

18. Forester, March 22,2006.

Vita

Kelly Thomas Howell was born in Lexington, Kentucky to Joyce Wathen and Thomas Lee Howell on November, 28th 1972. Kelly graduated with honors from the Science Academy at Lyndon Baines Johnson High School in Austin, Texas. Kelly received a Bachelor of Science degree in Mathematics along with a minor in Computer Science from Baylor University in Waco, Texas.

Kelly has worked at Computer Sciences Corporation in Austin, Texas for the last thirteen years providing consulting services for financial services clients around the world.

Kelly currently resides in the suburbs of Austin, Texas with his wife Lana and daughter Kathryn Ruth.

Permanent address: 1607 Purple Iris Cove, Pflugerville, Texas, 78600

This report was typed by Kelly Thomas Howell.