Mapping ACL/VF to a Java Environment: A Feasibility Study

By

Howard Scott Needham

A thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfilment of

the requirements for the degree of

Master of Computer Science

in

Computer Science

Carleton University

Ottawa, Ontario, Canada

August, 2015

Abstract

There are a number of different approaches to runtime validation, and one such method is Model Based Testing (MBT), a quality assurance technique where a test suite is generated from an abstract model. While there exists a number of different approaches to accomplish model-based testing, most are state-based (and thus faced with problems such as state explosion, correspondence to code, etc.). In this thesis, we instead focus on an alternative approach to MBT, namely scenario based testing, which we consider in the context of runtime validation. More specifically, we address scenario monitoring and validation in ACL/VF

ACL/VF. This approach, developed by Dr. Corriveau and his research team, provides both a language (ACL – Another Contract Language) to specify an implementation-independent testable model and a tool (VF – the Validation Framework) to validate an implementation against an ACL model.

The two current versions of the ACL/VF have a number of issues that prevent it from being a usable solution. The original version is .NET specific and incorporates external tools that are no longer supported. Upgrading it to a more recent version of .NET would amount to a complete rewrite. But, more importantly, this initial version has a major bug in its way of monitoring scenarios and a solution requires rethinking a significant and highly technical portion of the implementation of the VF. Instead a new version was implemented and tested with the JavaMOP framework. That solution requires manually mapping an ACL specification to a corresponding set of JavaMOP monitors. Experimentation with this second solution however revealed it cannot manage and monitor multiple scenarios running simultaneously!

In light of these very significant difficulties, the most immediate research question for ACL/VF is to determine whether it (and more specifically its rich scenario monitoring) can be implemented in a commonly-used environment such as the core Java/J2EE framework (ideally with no dependencies on external tools). This thesis provides an affirmative answer to this question: a case study is used to illustrate the proposed approach to monitor ACL scenarios using Java threads. We also provide a one to one mapping of ACL elements to corresponding Java/J2EE based code.

Acknowledgements

I wish to thank my supervisor, Dr. Jean-Pierre Corriveau for allowing me to work on this project. I would especially like to thank him for providing me direction through my research and offering alternative concepts and research direction, and last but not least, guiding me through and editing this dissertation. Dr. Corriveau has been an immense help and excellent mentor both in and out of the lab.

I would also like to thank my family for their encouragement and support over the past few years and putting up with my late nights and often absentee weekends. They have played an important role in my endeavors and without them I doubt I would be where I am today.

I would like to also thank my fellow lab members who have help direct me through my research and offered advice during the research phase. They have been there with me through all my trials and tribulations. It is great pleasure to work with them. Finally, I would like to thank my friends, both in and out of the university environment. They have been there to drag me out when I least expected I needed it. Reminding me to every once in a while to get away and leave the work behind and enjoy the friends I have.

Thank you one and all.

Table of Contents

Table of Appendices

Table of Figures

Definitions

context	Context refers to the current instance of the ACL contract
instance	Refers to the instance of the IUT class bound to a contract
Station Manager	The station manager is the ACL engine. It is responsible for the startup and maintenance of the lifecycle of the ACL/VF.
Monitor Station	A monitor station is responsible for the startup and configuration of contract monitors and the initialization of the related contract instance. It then controls their lifecycle
Contract Monitor	A contract monitor is responsible for initializing the IUT elements that are bound to the contract.
Main Scenario	A main scenario is defined as a time thread and is responsible for managing the temporal events associated with the scenario. A main scenario may trigger other secondary scenarios during its lifespan. It follows all the required semantics as defined in the ACL specifications with the exception of the termination condition.
Secondary Scenario	A secondary scenario follows all the semantics of a scenario as defined in the ACL specification, except that a secondary scenario must be triggered from within a main scenario.

List of Acronyms

ACL	Another Contract Language
CER	Contract Evaluation Report
IUT	IUT Implementation Under Test
CMM	Common Monitoring Model
SUT	System Under Test
TRM	Testable Requirements Model
TM	Traceability Matrix
VF	Validation Framework

1. Introduction

In the following chapter we define the problem addressed and provide a brief summary of the solution we propose. We first provide an overview of the problem and its context, namely the importance of testing specifications of an implementation under test (IUT) and how the ACL/VF provides an alternative approach to the current models that dominate the field of model-based testing. We then introduce the two versions of the ACL/VF that have been developed, and discuss the issues that prevent them from being practical solutions. Following this will be a description of our contribution and a more detailed explanation of why it is necessary. The chapter will then conclude with a summary of our solution and the methodology used to verify that it does indeed fulfill its promise.

1.1 The Context

It is widely accepted in the software industry that software quality assurance and testing (QAT) is a vital and integral part in any software development lifecycle (SDLC). In the majority of interactive design and development methodologies, (i.e. Agile Methodology, Unified Process (UP) or Rational Unified Process (RUP) [1] quality assurance and testing is an integral part of each of the SDLC phases (i.e. Inception, Elaboration, Development, Testing and Integration) and subsequent iterations. Although in one of his earlier papers (1970) Royce [2] expressed that quality assurance and testing was integral to the software development it was identified as a separate phase of the SDLC. In 1988 in a paper on software development and the spiral model Boehm [3] introduced the need for validation and testing in each of the SDLC's and has since become common practice.

One of the primary purposes of software testing is to ensure that the IUT conforms to the requirements defined by the stakeholders and does so with minimal fault [4]. These are generally

in the form of software requirements specifications (SRS), which rest on requirements engineering (RE) [5, 6, 7, 8]. More specifically, the SRS must be validated against the behavior of the IUT [9]. The RE process focuses on capturing both functional and non-functional requirements. While functional requirements (FR) define the specific tasks and behavior of the system, non-functional requirements (NFR) represent the quality attributes of the system. In short, FR define "how the system is supposed to work" and NFR define how the system is supposed to be. The capturing and implementation of both types of requirements is vital from the stakeholder's point of view and as such both functional and non-functional requirements must be included in the validation process [10].

One of the critical issues in QAT is that the validation process must verify the IUT addresses and meets each of the stakeholder's requirements [11, 12]. Central to this issue is traceability. Traceability concerns every aspect of the RE process and must ensure there is a mechanism in place that both traces and records the life cycle and decisions made by the principal artifacts of the IUT. Both the artifacts and decisions must be mapped, or traced back to each of the stakeholder's requirements [12, 13].

1.1.1 Approaches to Validation

There are two primary approaches to the validation process, code-centric and model centric. Code-centric approaches to software testing, such as Test-Driven Design (TDD) and Agile Development focus primarily on test cases, or unit tests written at the implementation level and drive development [14, 15]. TDD and Agile Development focus on the fast development of working code and are based on a software development processes that supports iterative and incremental development. In this approach design and testing are integral to the development process. This technique focuses more on developing software that meets the current needs of the

client rather than attempt to design flexible and reusable solutions. In effect, developers pursue the simplest solution that satisfies the stakeholder's needs.

The unit test approach is reflective of white box testing in that the individual components of the system are tested in isolation to determine they function properly without consideration to the internal structure and interaction of the system (i.e. black-box testing) [16, 17]. A common framework that drives these tests is the XUnit framework, for example JUnit for Java centric applications, and CppUnit for C/C++ centric applications [18]. As such the production of high quality code rests ultimately with the developer [19, 20].

Such design and development processes leave little to no documentation or artifacts behind and as such traceability may be limited to the current set of test cases. With the focus on rapid design and development and limited documentation code-centric approaches may become marred with implementation details and as such may become difficult for the stakeholders to understand and use [11]. Due to the lack of formal specifications and models this form of design and development may have a negative impact on integration and acceptance testing and may hinder future maintenance of the application. More on this in Chapter Two.

In model-driven design and development (MDD) the needs of the stakeholders' are expressed as models and from these models tests are extracted and used to drive the validation process [21, 22]. The concepts and principles of MDD are not new. Model driven design and development is one of the basic principles in science and engineering that dates back to 1697 when Sir Isaac Newton first published his works Principia [23]. In software engineering (SE) MDD uses a variety of system independent graphics and textural languages to capture the software requirements. These models present a view of the system that is abstract and usually independent of the implementation details and as such may be more readily understood by the stakeholder's [11, 24, 25, 26]. The

models generated serve as input and output during all stages of the SDLC, including integration and acceptance testing. During the development lifecycle these models are fluid in nature and are rigorously tested, updated, tested again until the system meets the stakeholder's requirements [26, 27, 28]. The abstract nature of MDD lends itself to model-based testing (MBT). MBT generates test suites from abstract models and captures a platform independent representation of the implementations behavior. The use of platform independent test suites provides two significant advantages, the first is it decouples requirements from the IUT, and the second is the reuse of test suites across multiple IUT's [24, 29, 30].

1.1.2 Model Driven Testing and Use Cases

One variation of MDD that has gained popularity over the past two decades is the Object Management Group's Model-Driven Architecture (MDA). The MDA models the stakeholder's requirements using OMG's Unified Modelling Language (UML) [31, 32]. UML is a platform independent and textural based language that is used to capture both the functional and non-functional requirements. One of the key features of MDA is that it must support incremental and iterative development. In general this means that mappings between models must be repeatable and refineable; however they must not be integrated into the source model [33]. Although UML has a plethora of artifacts used capture the design and implementation of the system under development (e.g., Class Diagrams, Sequence Diagrams, Activity and Component diagrams, etc.), Use Case Diagrams constitute the principal artifact employed to capture user requirements [34, 35].

Fundamentally a use case is a specification of actions, including variants that a system performs while interacting with external actors of the system. In UML an actor is typically any external agent that interacts with the system such as a user, database or other computer systems or

applications. A use case defines a systematic way of employing the system to perform a specific function. A use case instance, also known as a use case scenario is a sequence or set of events and actions defined in a use case that is carried out under specific conditions [36, 37]. Each use case may capture several scenarios each corresponding to a possible path through the use case [38, 39]. Implicitly, use cases and their scenarios capture the requirements of the stakeholders' and define the responsibilities (i.e., fine grained requirements) of the system [40, 41]. In addition, a use case captures both the pre-conditions and post-conditions of the scenario itself. The use of use cases and scenarios as a method for requirements gathering is commonly accepted and, due to their intuitive nature are generally preferred by the stakeholders' [42, 43] to state-based specifications. More on this is Chapter Two.

When discussing Use Cases we must mention two important extensions for which there exists considerable literature: extended use cases (eUC) and use case maps (UCM's). First, In 2000 Robert Binder introduced the concept of an extended use case for system level testing [44]. As Ul Haq defines in his thesis, an eUC can be thought of as a parameterized use case where the parameters of each use case corresponds to the path sensitization variables required to cover the different paths or scenarios through the use case [40]. Second, in 1996 Buhr and Casselman introduced Use Case Maps in their book Use Case Maps (UCM's) for Object Oriented Systems [45]. Use case maps provide a visual notation for use cases and also a means of extending them into high-level design. UCM's can be modeled to express different paths within the same scenario and identify different branches and merging conditions. However, understanding use case maps does not depend on familiarity with use cases: Use cases and use case maps may be viewed as an independent yet collaborative methods of defining scenarios and their related paths, more on this is Chapter Two.

1.1.3 Event Driven Design and Testing

Event Driven Design and Testing (EDDT) [46] is another development and testing technique relevant to our thesis. Event driven systems are designed to process events as they occur. This allows the system to observe and respond dynamically to these events as they are generated. EDDT may be thought of in terms of the observer pattern where there is a one to many relationship between objects: when one object changes state (the subject), all its dependents (the observers) are notified and systematically updated. The event issues specialized data that is dependent on the event that occurred and the recipient that is responsible for processing it. In addition, events may be identified and linked together to form event-driven process chains (EPC's). These EPC's encapsulate business logic that is used to identify critical areas in the process model that controls the flow of system [47, 48, 49]. This is relevant to use case scenarios and path sensitization and will be explained in further detail in Chapter Two.

1.1.4 Another Contract Language and the Validation Framework (ACL/VF)

Based on the design-by-contract paradigm ACL/VF is a validation framework that incorporates model-based and scenario-based testing. Developed by Dave Arnold and Dr. Corriveau ACL is built on two well-known and established principles, 1) expression of requirements in terms of responsibilities and scenarios and 2) organization of these responsibilities and scenarios into contracts [50]. Contracts in ACL are specifications of the IUT types and are bound to these types to run static checks and runtime verification. This binding to types occurs prior to runtime and consists of associating each contract to an IUT type. At runtime, instances of the contracts are created and bound to their respective IUT type instances. The ACL/VF executes the ACL contracts alongside the IUT and relevant information is exchanged between the two executions [51, 52, 53]. More on this in Chapter Three.

Contracts consist of several different components including scenarios and responsibilities. While contracts make use of both static and dynamic checks, runtime monitoring is conducted through scenario monitoring. Static checks are performed before the execution of the IUT and look at the system's structure, dynamic checks are executed at runtime and typically address system behavior [51, 52, 53].

The majority of the ACL validation process occurs at runtime and requires one additional input – namely a set of bindings. Bindings map the classes of the IUT to the contracts they must obey, and each responsibility identified in a contract must be mapped to either an attribute or a procedure in the IUT. Each time an instance of a class is created a corresponding instance of the contract is created. In addition ACL incorporates pre and post condition checks, invariants and event driven monitoring to observe and monitor scenarios [11, 24, 40].

1.2 Problem and Contribution

Central to ACL is the ability for the VF to distinguish between multiple instances of a single class executing concurrently during scenario monitoring. In addition the ACL/VF framework must provide an engine, whether manual or automatic that binds the ACL Contract to the IUT. Currently there are two versions of the ACL/VF and a number of issues inherent to their design make them as unusable.

1.2.1 ACL/VF .NET Version

The first version was developed in Microsoft's .NET 3.5 environment. After extensive experimentation a number of issues were identified that that prevented it from being a usable solution. First, is the inability of the ACL/VF to distinguish properly between multiple instances of the same class executing concurrently during scenario monitoring. Second, the tools and mechanisms used within the ACL/VF to bind the contract to the IUT are extremely complex and

.NET specific [54]. Third, the ACL/VF was dependent upon an external tool, specifically Microsoft Phoenix, to a) capture the required IUT binding information as well as static checks, and b) programmable breakpoints to pause the IUT execution for checking against the testable requirements model (TRM) [54]. In addition, the VF does not support the monitoring of atomic statements and parallel blocks. The inherent complexity and dependency on the .NET 3.5 environment and Microsoft Phoenix led the research team to determine that any future work on the original VF to handle newer versions of the .NET framework would have to be scrapped completely and redesigned in order to adequately update the ACL/VF.

To this end Dr. Corriveau and his research team decided to conduct a study on whether the migration of the ACL/VF to Java was feasible. Java was chosen due its widespread usage and platform independence. The directive was to identify current technologies and tools that would support the migration of the ACL/VF to the Java language framework while adhering to as much of the ACL syntax and semantics as possible.

1.2.2 JavaMOP and JUnit Version

After extensive research two primary tools held significant promise, JUnit and the JavaMOP framework. From these findings a solution was proposed that utilized the JavaMOP framework to generate AspectJ monitors for the runtime monitoring of a Java IUT, with specific focus on scenario monitoring, and JUnit to test the pre and post-condition and invariants throughout the execution of the IUT [24]. The JavaMOP and JUnit framework will be discussed further in Chapter Two.

However, after extensive testing inherent issues arose that prevented this from being a usable solution. First, JavaMOP is developed and supported by a research team at the University of Illinois. Future enhancements to the ACL/VF would be reliant upon future releases of JavaMOP.

As such future releases of JavaMOP may not address our issues directly. The latest release of JavaMOP compiler was version 4.2 issued July. 2015. However the previous release 4.1 was issued on Nov. 2012 [55]. This issue is reminiscent of the original ACL/VF version with respect to limited support of external tools. Second, while JavaMOP supports RT-Monitoring, it is limited to monitoring a single instance of the IUT and the set of related ACL contracts. In essence, only one set of contracts can be initialized and tested against the IUT at runtime. Third, runtime monitoring of parallel blocks was not implemented. In addition, JavaMOP is a complex language with a steep learning curve and both the contract monitors and scenario monitors have to be coded from scratch.

1.2.3 Our Contribution

Faced with these four issues namely, 1) limited support and extensibility of JavaMOP, 2) limitation to a single instance of the contracts under runtime verification, 3) the lack of support for parallel blocks, and 4) JavaMOP/AspectJ coding requirements, Dr. Corriveau and his research team looked to determine other feasible solutions.

Learning from previous research four fundamental principles emerged. First, migration to Java and the related Java framework appeared to be a feasible solution to the original problem. Second, the implementation of monitors provided the robustness needed to monitor multiple instances of an IUT running concurrently. Although JavaMOP did not support this aspect of the VF requirements, the concepts and principles provided an attractive solution for runtime monitoring. Third, event monitoring and the identification of cross-cuts[1] provided a solution to monitoring multiples instances of the same scenario running simultaneously. And fourth, the implementation of parameterized events is essential for complex scenario modeling. Similar to extended use cases,

[1] In aspect-oriented software development, **cross-cutting concerns** are aspects of a program that affect other concerns in separate classes. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and can result in either scattering the responsibilities or functionality (code duplications), tangling (tightly coupled systems where there is significant dependencies between systems), or both.

parameterized events include variables that can be used for runtime checks and verification of the temporal ordering of events. This will be discussed in detail in Chapter Three.

The key question became: is it feasible support ACL syntax and semantics, especially its use of scenarios and atomic and parallel blocks, in a pure Java based environment, free of any external tools. The contribution of this thesis is to provide an affirmative answer to this question.

1.3 Overview of Methodology and Solution

ACL is a high-level specification language that is closely tied to the concept of contracts and responsibilities and offers several constructs that support system level monitoring, namely scenarios and responsibilities [56]. In addition we must note that ACL is a textual based language that is implementation independent. In this section we will briefly identify and discuss the key ACL/VF elements and functionality we address and

1.3.1 Key ACL Elements

At the core of ACL is the concept of the contract. Contracts are a specification tool that defines the behavior of a system or IUT. Contracts describe conditions that must be true (pre-condition) before a method is called and the conditions that must be true after the method is called (post-condition), providing the method is called correctly. In addition, contracts describe object invariants or properties that must hold whenever the object is visible [57].

Along with other constructs, embedded in the contract are scenarios and responsibilities. Scenarios in ACL are closely tied to use cases and use case scenarios. A scenario may be considered a single path through a use case and as such, depending on the variants identified, may have multiple executable paths [58, 59]. In ACL a scenario defines the grammar that contains a sequence of events and responsibilities that is used to monitor a specific function of the system.

A critical element of ACL is the responsibility, and a contract may contain multiple responsibilities. Each responsibility identifies a requirement or task that has to occur along with the dynamic checks associated with the responsibility. A responsibility may be used in one of two ways. First, a responsibility is bound to a method or group of methods in the IUT. The second, an unbound responsibility, is through the use of execution grammar that may in turn implement other responsibilities to define its task [60], as will be illustrated later.

Another key element is the ACL event. An ACL event is fired by a responsibility or a scenario and is global to all instances of the contracts executing at the time. An event is a communication mechanism between instances of contracts that indicates the current state of the instance at the time the event was fired. Contract variables are another key element. Contract variables exist within the context of a contract and have the same lifespan as the instance of the contract containing them. In summary: scenarios define the temporal grammar, responsibilities define the behavior of the IUT during runtime, and events are the communication mechanism between instances of the contracts running concurrently. Essentially, scenarios are used to drive the runtime verification of the ACL/VF tool [24]. The input is an event that may be time dependent and the output is a set of reports that can reviewed to validate if the sequence of events defined by the scenarios temporal grammar has occurred in the correct sequence and the corresponding paths and events occurred between interdependent contracts in the defined order.

1.3.2 Key ACL Concepts and Functionality

There are a number of significant aspects of contracts and scenarios in ACL that are central to our thesis. We will provide a brief overview and discuss them further in Chapter Two.

First, a contract may have several instances running concurrently, each with their own set of instance-bound contract variables. A key issue is that the ACL/VF tool must be able to distinguish

between these instances, and as such must be able to monitor the runtime of each contract instance simultaneously. That is, upon the triggering of an event or the occurrence of a responsibility, the tool must be able to distinguish between instances and identify the corresponding contracts. This functionality is missing from both the original VF and the JavaMOP version [24] and must be addressed directly in our proposal.

Second are contract scenarios. Recall: a contract may have one or more scenarios each with their own set of responsibilities and related events. A scenario defines the temporal grammar that the responsibilities and events must honor in order to be considered valid. During runtime verification there may be one or more instances of a contract running concurrently, and each instance of the contract may have one or more scenarios running simultaneously. Furthermore, each scenario has its own variables whose values define a specific path through the scenario. In brief, there can be multiple instances of a contract running simultaneously, each with a number of scenarios running simultaneously, and each of these scenario may have multiple paths, with each path defined by the possible values of its variables. The proper handling of the various scenarios and related atomic and parallel blocks that may exist simultaneously is not trivial and requires serious consideration.

Third, we consider event monitoring. Recall events may be considered the communication link between contracts and their scenarios. Events are used to trigger the next responsibility or sequence of events. As such ACL events may be considered global in nature and contract independent. This requires that events are monitored and processed by all related instances of the contracts running simultaneously during runtime.

Fourth, we consider contract and scenario monitoring. Currently both the contract monitors and scenario monitors have to be coded from scratch. This requires the mapping of ACL contracts

to Java based classes be coded from scratch and as such may require specialized skills, specifically JavaMOP and AspectJ programming.

Finally, as previously mentioned, the proposed ACL/VF should be platform independent and be independent of any external tools. This will allow for the flexibility of future development and maintenance of the ACL/FV.

1.3.3 Overview of Our Solution

Based on previous work, four primary directives emerged: 1) determine if ACL/VF, in particular its scenario definition and runtime monitoring can be implemented in a commonly used environment, i.e. Java, ideally with no dependencies on external tools, 2) develop a solution that maps ACL contracts and its code readily to a set of Java based classes, 3) eliminate the need to write scenario monitoring code, and 4) demonstrate the feasibility of supporting ACL/FV with this tool or framework. This thesis provides a solution that 1) addressed the issue of runtime monitoring of contracts and their related scenarios, including atomic statements and parallel blocks through the use of Java and Java threads, 2) maps the ACL grammar to Java/J2EE based classes, and 3) through the use of a case study demonstrate the proposal is not only feasible but constitutes the basis for a full implementation of ACL/VF in the Java/J2EE framework. Our work is based on a well-known system-level case study "the University Example" developed by Dr. Corriveau and Dave Arnold [61].

In summary, our solution encapsulates the scenario monitoring code in the ACL/VF and the user no longer has to write the code from scratch. This allows the user to reuse the support classes across multiple contracts without being concerned about writing the related monitoring code from scratch. In addition the related Java code required to map the ACL contracts to their corresponding

Java based classes is very close to the ACL scenarios they will have written in the first place, minimizing the mapping complexity.

In order to accommodate our solution we have modified two ACL types and introduced three new ACL types, namely, 1) modifications to the parallel block, 2) revised the choice statement, 3) the introduction of parameterized events, 4) the introduction of a time sensitive scenario, the TimeThread and 5) a new ACL type – the Dictionary. In order to accommodate our recommendations we propose a newer version of ACL be considered. These will be discussed in detail in both Chapters Three and Four.

1.4 Overview of the Thesis

Chapter Two is the literature review. We discuss the current technologies and approaches that are relevant to our work. Chapter Three identifies the current issues ACL/VF faces with respect to scenario monitoring and provides a detailed discussion of our solution. Chapter Four presents the University Example and discusses the strategies we developed for scenario monitoring, namely Java and Java threads. The goal of this chapter is to explain how we developed the strategy for mapping ACL and the Validation Framework to the Java/J2EE framework. The mappings discussed in Chapter Four will become the basis for ongoing efforts to generate Java code from ACL contracts. Chapter Five summarizes our work outside the context of the example provided in Chapter Three. Finally in Chapter Six we will present our conclusion and identify future work in this area.

2. Background

This chapter presents a review of work related to the topic of our thesis. First we provide a brief review of the model-based and event-driven testing approaches. We then briefly discuss extended use cases (eUC) as proposed by Binder (200) and their application to scenario based testing approaches and ACL.

2.1 Model Based Testing

As discussed in Chapter One, state-based MBT tools have a number of inherent issues and consist of problems such as test selection, test management and state explosion, the most prominent being state explosion [62].

The basic promise behind MBT is that from a formal or semi-formal model (e.g. transition system, UML State Machines, class diagrams extended with constraints, etc.) a complete set of test cases (input and expected output pairs) can be generated. However, MBT is known to readily generate a lot of test data even from small models. This is commonly referred to as the state-explosion problem. The problem stems from an approach that defines the state of the system as a set of values (i.e. instance variables or equivalent data members) held by the objects of the system at specific times during the validation process [62, 63]. In a traditional state-based system, the state-based or combinatorial explosion comes from the combination of the number of variables and possible values for each variable [25]. However, more recent work on MBT takes an approach where the states are not defined by the variables of the objects but that each state may pertain to a sequence of methods or events [38, 62]. This can be seen in tools such as Microsoft's Spec Explorer [62] and temporal logic approaches [64, 65, 66] where the temporal logic formula is translated into state machines [67, 24]. More specifically, in Spec Explorer state explosion is compounded by the by the possible sequence of method calls as well as the valid set of parameters

for those method calls. This is also the case with temporal logic approaches that parameterize the logic [24, 68]. In effect, approaches that do not define parameters have no way of verifying whether a valid or invalid sequence of events or method calls has occurred since parameter values have to be checked [24].

2.2 Event Driven Software and Testing

In Chapter One we introduced event-driven software (EDS). In EDS messages are sent between components and the components react by changing their internal state, respond with messages (or events) and/or wait for the next message or event to occur. In EDS applications the software does not execute a set of instructions in a predefined or scheduled sequence. On the contrary, users control the flow the program through various operations and through the use of event-handlers[2] the inter-program instructions are called by an event triggered by the user (e.g. mouse click).

Event driven methods emphasize the analysis and design of the system around event initiation and management and they don't manage events according to schedules or sequences. In effect, the program avoids sequence management and become more flexible [46, 69]. Therefor we must consider that characteristics of the event and the mechanisms that both trigger and consume them. There are six criteria that event-driven testing must consider [46].

[2] Event handlers are program code that handles or responds to an event. Through the development of a set of API's event-handlers be developed and maintained independent of the application software. This separation of concern (or decoupling) allows complex system to be built using a collection of loosely coupled pieces of code. In addition, through the use of event-handlers, the user is not restricted to a fixed order or sequence of input [70].

Criterion 1: Event coverage criterion

There should be enough test cases generated such that each event is executed or fired at least once. That is, the trigger condition for each event identified in the system should be satisfied at least once.

Criterion 2: Event-statement coverage criterion

Sufficient test cases must be generated such that each statement in the program is executed at least once.

Criterion 3: Event-decision coverage criterion

There must be sufficient test cases generated such that a *true* or *false* value of each decision in the program of each event is executed and recorded at least once.

Criterion 4: Event-condition criterion

Similar to criterion 3, there must be sufficient test cases generated such that a *true* or *false* value of each condition in the program of each event is executed and recorded at least once.

Criterion 5: Event-decision/condition coverage criterion

This criterion covers the interaction between criterions 3 and 4. In effect, there must be a complete set of test cases such that a *true* or *false* value is generated and recorded for each decision expression in the program a least once and for each possible value of each internal condition.

Criterion 6: Event-condition combined coverage criterion

These test cases cover the various possible value combinations of each decision expression of the program of each event.

Although event coverage criteria is essential for event driven testing it is not sufficient enough to address the dynamics of an EDS during runtime verification. During runtime verification event

execution cannot be predetermined as the events are not always executed by predefined sequences and as such different event sequences may bring different results. So, in order to consider event execution sequences the event execution rules need to be studied and identified.

In addition, like MBT EDT has their own inherent issues. These come about primarily because of the large number of combinations and permutations of events that need to be handled by the EDS. Since, in principle, events may be executed in any order, unanticipated interactions between them lead may lead to software failures. The large number of interactions may also create state explosion similar to that of MBT. An additional challenge is that very few specialized techniques exist to test EDS. There is little tool and artifact support for an EDT environment [70].

2.3 Scenario Based Testing

Scenarios, which are generally extracted from use cases, are used to describe the functionality and behavior of a software system in a user-centered perspective. Scenarios form a kind of high-level set of test cases for the system under development (SUD) [42]. Basically scenarios describe a set or series of responsibilities where each responsibility represents a simple action or task taken by the IUT. In a scenario based testing approach, IUT-independent tests are developed from the scenarios and transformed into test cases that can be run against an IUT. These test cases correspond to a set of paths or transversals through the scenarios. The question becomes: given a set of scenarios which specific paths must be tested [71] and to what degree and level of testing [53, 52].

While there has been considerable research and development of tools develop for utilizing scenario in test development, especially in the field of model-based testing, our thesis will focus primarily on scenario based testing and its application to the ACL/FV. Further discussion of

scenarios and scenario monitoring will be presented in Chapter Three where we present the ACL Validation Framework and monitoring requirements in further detail.

2.4 Extended Use Cases and Scenario Based Testing

In Chapter One we briefly discussed use cases and how their scenarios capture the requirements of the stakeholder and define the responsibilities of the system. It is important to note that each use case may in fact capture several scenarios; with each scenario corresponding to a possible path (or traversal) through the use case. In addition, use cases captures the pre-conditions and post-conditions of the scenarios themselves [71]. However, it is widely acknowledged that UML's support for modeling inter-use-case semantics is very limited. In 2000 Robert Binder introduced the concept of an extended use case for system level testing [44].

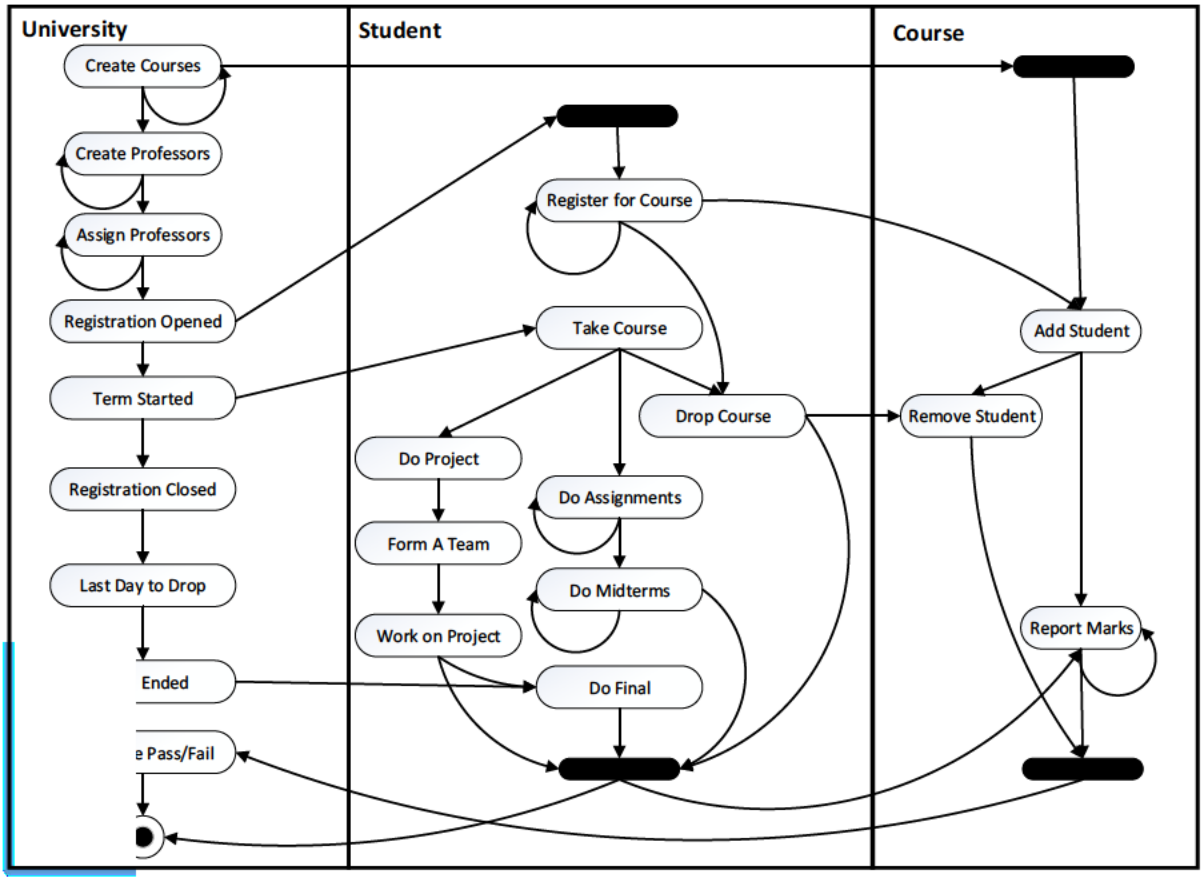

Figure 1: University Example and Use Case Interleaving

As Ul Haq defines in his thesis, an eUC can be thought of as a parameterized use case where the parameters of each use case corresponds to the path sensitization variables required to cover the different paths or scenarios through the use case [40]. These parameters allow for the modeling of the sequential dependences and *interleaving*[3] between use cases. However, the notion of interleaving, presents one major problem: even for small examples, the number of sequences of use cases to test is essentially unmanageable. This is demonstrated in the previous figure.

You will quickly notice the interdependencies between the use cases. For example, both the Course and Student use cases have pre-conditions that are dependent upon elements of the University use case; i.e. the Course use case has the Universities create courses element as a dependency, the Students use case has both the Universities registration opened and term started elements as a pre-conditions. In addition, the University use case has a dependency on a post-condition of the Course use case. You will also notice how quickly these sequential dependencies may lead to the number of test cases generated becoming unmanageable. For example a single university has multiple courses and multiple students, with each student registering in one to many courses and each course having one to many students registered in it[4]. This is similar in nature to the state explosion found in both the MBT and EDT paradigms. Since scenario based testing is driven by the possible paths or transversals identified by use cases this presents a critical issue for scenario driven testing.

[3] Interleaving can understood as cutting a use case interaction into fine grained pieces and weaving them into the interaction description of use cases. Any interleaving is possible as long as the relative order of the use case elements is maintained [71].

[4] The calculation of the number of paths or transversals is outside the scope of this thesis. The algorithms and computation to determine the number of test cases are provided in Briand and Labiche paper [108].

2.5 Use Case Maps and Scenario Based Testing

Another topic briefly discussed in Chapter One that is relevant to our thesis is use case maps (UCM) and their relationship to scenario based testing. As mentioned in Chapter One, use case maps provide a visual notation for use cases and also a means of extending them into high-level design. UCM's can be modeled to express different paths within the same scenario and identify different branches and merging conditions. However, understanding use case maps does not depend on familiarity with use cases: Use cases and use case maps may be viewed as an independent yet collaborative methods of defining scenarios and their related paths or transversals [45, 72, 73].

UCM's link the behavior and the structure of a system in an explicit and visual way. The UCM paths are a first-class architectural entities that describe the causal relations between responsibilities [73]. When discussing UCM's there are two types, unbound and bound. An unbound UCM describes high-level scenarios in terms of causal relationships between responsibilities while not identifying the system level components they are bound to. On the other hand, a bound UCM identifies the underlying structure or system components the UCM's path transverse and triggers the responsibilities. In essence, a *bound use case map* is layered on top of a set of system components that the responsibilities are bound to [74, 75].

While it is beyond the scope of this thesis to present a complete review of UCM artifacts, we will discuss those elements that are most relevant to our thesis. A use case map consists of a start point (filled circles representing pre-conditions), responsibilities which are represented by crosses, and end-points (post-conditions) that are represented by bars. As previously mentioned, responsibilities can be bound to components which are the entities or objects the system is composed of [73, 74, 45, 76]. In addition, Buhr and Casselman noted that real-time may enter the

map explicitly (pg. 41). In order to accommodate this they introduced a timer element[5]. A timer indicates the existence of a time delay, but not its details. A timer may be thought of as a special type of responsibility along a path that takes up real time without taking up processing resources [45, 77, 78].

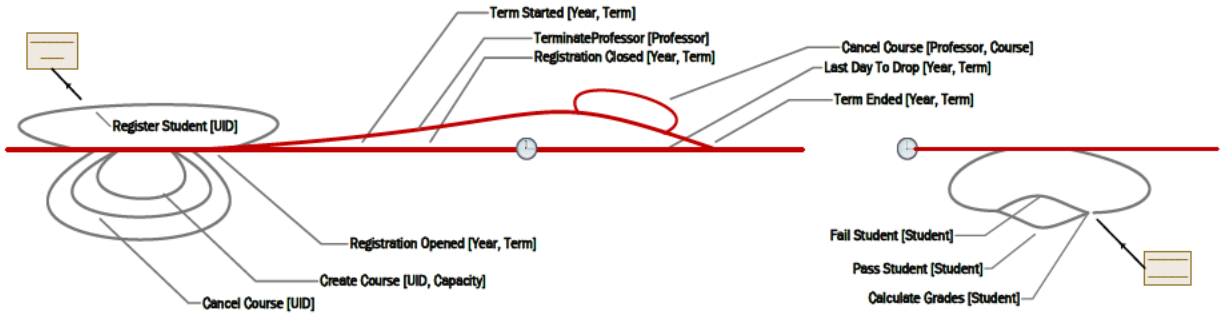

Figure 2: Example of a Use Case Map from the University System

The wiggly lines are the paths, or scenario paths that connect the start points, responsibilities and end points. The arrow heads indicate the direction of the path and any divergence and/or convergence that may occur along the path. Although not identified as such, branches may be analogous to decision points or alternative flows found in the use case scenarios. There is also a symbol introduced by Buhr and Casselman that identifies a pool (square box with an outgoing arrow). A pool is a place for holding dynamic components that are ready to move into a slot. A slot is analogous to a responsibilities in that they have fixed positions and fixed responsibilities along the paths that transverse them.

Another key aspect to UCM's that is relative to our thesis is that a UCM may represent multiple paths that a scenario may execute concurrently (represented by a fork).

[5] Buhr and Casselman also refer to use case paths as "timethreads" (pg. 14) in that they both identify the same concept [45].

Along with the ability to introduce time dependent elements in use case maps, or timed use case maps[6] (TUCM) [77, 78] UCM may be used to derive Message Sequence Charts (MSC). MSC may be used to develop and present the details of interactions between components of the system [79].

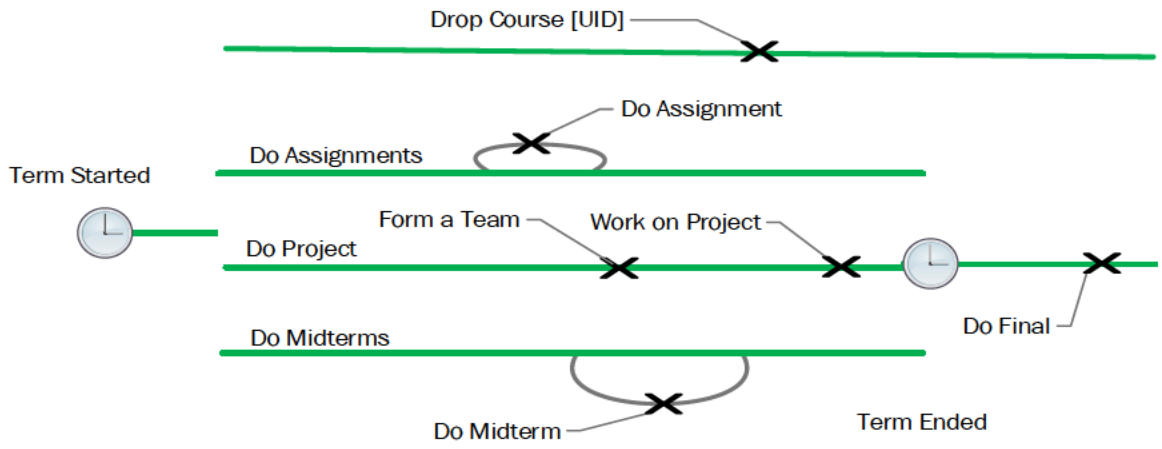

Figure 3: Example of a Use Case Map with Concurrent Paths

As Miga et al. (2001) point out [79], UCM's remain stable over changes related to messages, protocols and communication constraints with respect to the underlying structure of the system. As such UCM's remain focused on the intended functionality and on the reusable causal scenarios (with respect to messages). With this in mind, UCM's may be used to develop the MSC between scenarios and further refine the message sequencing and communication constraints.

2.6 Another Contracting Language (ACL)

As introduced in Chapter One, ACL is a high-level, implementation-independent specification language that is based on the design by contract paradigm. Put forth by Arnold and Corriveau [51, 53, 52] the goal of ACL is to provide a new approach to a testable requirements model (TRM).

[6] While detailed discussion of Timed Use Case Maps (TUCM) is beyond the scope of this thesis it is important to note that TUCM are becoming commonly accepted for developing user requirements and modeling system behavior that have temporal or time dependent elements. For a detailed discussion of TUCM and developing the timing constraints see Hassine et al. (2006) paper [78].

ACL is a textural specification language that is intended to be implemented after the user requirements notation (URN) specifications have been developed. URN combines two existing notations, goal-orientated requirements language (GRL) and use case maps [80]. More specifically, ACL is intended to be written based on the refinements of the responsibilities, scenarios, and metrics defined by the GRL and UCM diagrams (developed from use cases) [53].

In this section, we summarize the essential aspects of ACL that are relevant to our thesis and provide examples of their usage through examples from the university project [82]. All pages numbers provided at the end of each section is in direct reference to the ACL specifications Version 3.1 [80].

2.6.1 Contracts

One of two central elements in ACL is the contract. A contract is bound to a type. However, the type is not defined in the ACL but is actually a part of the IUT. A contract defines the functional requirements of a type as well as the scenarios in which the functional requirements may occur. A contact may contain several elements that describe the responsibilities, observabilities, scenarios, static and dynamic checks, metric evaluators and binding rules. These elements are discussed later in this section.

ACL defines two types of contracts; main and regular. In addition a contract may be defined as abstract. An abstract contract (similar in nature to an abstract class) contains information common for all contracts that inherit from it. An abstract contract cannot be used as a complete contract in itself. Inside an abstract contract any ACL construct can be declared abstract by preceding the construct with the reserved word *abstract*. Concrete definitions for the construct must be defined in the subcontracts (the contracts that inherit from it). An abstract contract is not bound to and IUT type.

```
MainContract University <University>

Contract Student <Student>

abstract Contract ContainerBase<Type T>
```

Example of a Contract Definitions

Regular contracts are declared using the Contract keyword and main contracts are declared using the MainContract keyword. Both contracts may contain the same element types and follow the same ACL grammar. The only difference between the two is that a main contract is automatically bound to an IUT type at compile time while regular contracts are only bound to an IUT counterpart when used. In other words, regular contracts are only bound if used or referenced within the current contract project (pg. 66-72). This is considered dynamic or runtime binding.

2.6.2 Responsibilities

The second of the two central elements to ACL is the responsibility. A contract may contain any number of responsibilities. A responsibility can be considered a functional requirement or task. Responsibilities are classified as unsafe methods in that they can make changes to aspects of the IUT under runtime (such as variables). A responsibility can contain any number of parameters, pre and post-conditions, and beliefs[7].

```
// Type = bound
Responsibility tCourse SelectCourse(List tCourse courses)
{
      Execute();
      Post(value not= null);
      Post(courses.Contains(value) == true);
}
// Type = unbound
Responsibility DoProject(tCourse c)
{
      FormATeam(c),
      observe(TeamFinalized),
      WorkOnProject(c);
}
```

[7] Although beliefs are not central to our thesis they deserve a brief mention. In ACL beliefs are used to provide user friendly names and descriptions to a set of checks. This will be discussed further in Chapter Four when we discuss asserts.

Example of a bound and unbound responsibility

There are two types of responsibilities, bound and unbound. The first type, or a *bound responsibility* is bound to a method or group of methods found in the IUT. The second way, or an *unbound responsibility* is to define the responsibility using execution grammar that in turn uses other responsibilities. All bound responsibilities must contain an Execute () statement that shows when the method in the IUT is bound.

There are two special types of responsibilities, *new* and *finalize*. Both of these methods contain no return type or parameters. The body of the "new" responsibility is executed immediately following the creation of a new contract instance. Similarly, the body of the "finalize" responsibility is executed immediately before the destruction of the current contract instance

```
Responsibility new()
{
      Post(Courses().Length() == 0);
      Post(Students().Length() == 0);
}
Responsibility finalize()
{
      Pre(Courses().Length() == 0);
      Pre(Students().Length() == 0);
}
```

Example of a new and finalize responsibility

One aspect responsibilities not covered in our test case is that a responsibility in ACL, like an observability, can be preceded by a statement that determines if it is to be included in the contract at runtime. In this case, the statement would look like this:

```
[Parameters.DestroyProfessor == true] Responsibility DestroyProfessor (T professor)
```

This indicates that the responsibility, *DestroyProfessor* would be included only if the parameter in the contracts parameter block is set to true. This allows contracts to be broken up into components that are included or excluded at runtime based on the value set in the Parameters block

(pg. 144-122). This aspect of a responsibility helps in writing contracts in the Generative Programming[8] style.

2.6.3 Observabilities

Observabilites are another important element of ACL. Like responsibilities a contract may contain any number of observabilities. Observabilities are read only methods and are used to aquire state information from the IUT for evalutation (during runtime validation) within the contract and only funtion when the IUT is being executed. Again, like responsibilties there are two types of observabilities, bound and unbound and are similar in nature as their responsibility counterpart. In effect, they can be bound to an IUT method or attribute, or unbound, where they are used to compute a value based on some contract variables. An observability can contain any number of parameters, pre, post and check conditions

```
Observability List<tCourse> CurrentCourses();
Observability List<tCourse> CompletedCourses();

Observability String Name();
Observability Integer StudentNumber();

Observability Boolean Cancelled();
Observability Boolean IsFullTime();
```

Example of bound Observabilities

An observablity must return a value. If the body is executed as a single statement or expression, it is the value of the single statement or expression that used as the return statement. Otherwise a value statement must be specified. A value statement is similar to an assignment statement and it is the value that is returned to the caller (pg. 104-109).

[8] Generative programming is a style of computer programming that uses automated source code creation through generic frames, classes, prototypes, templates, aspects, and code generators. Aspect programming, the use of assertions, Java annotations and Java Generics also fall into this category. This is covered in detail in Chapters Three and Four

```
Observability Boolean IsCreated()
{
      StudentNumber() > 0;
}

Observability Boolean DroppedCourse(tCourse course)
{
      observe (not LastDayToDrop);
      value = instance.DroppedCourse (course);
}

Observability Boolean AssignmentCompleted(tCourse course)
{
      Boolean value;
      Pre (Assignments().ContainsKey(course));
      Check(Assignments().Length() > 0);

      loop (1..Assignments().Length())
            value = value && Assignments(iterator).IsCompleted());
}
```

Example of unbound Observabilities

An observability may also contain an observe element. An observe element indicates that execution does not continue until the observable event, identified by the given identifier is observed. Any abstract observabilities defined in an abstract contract must be refined in a derived contract.

2.6.4 Parameters and Variables

A contract may contain zero or more parameters. The paramaters section, is usually the first section of the contract that may be used to configure the contract. A parameter definition is similar to a variable declaration, except parameter values are specified before IUT is executed. Parameters can be used to specify different scenario paths, repitition constants, or to define a constant value, such as the maximim number of students in a class (pg. 88).

The value in which the variable is set to can be done in one of two ways. The first way is for it to be set directly in the contract. The second way is for it to be set by the binding tool. The second way is for the parameter variable to be set at runtime. This is denoted by the InstanceBind keyword.

```
once Scalar tUniversity instance;

Parameters
{
   [1 to 12]  Scalar Integer InstanceBind NumTermsToComplete;
   [1 to 100] Scalar Integer InstanceBind UniversityCourses;
   [1 to 100] Scalar Integer InstanceBind UniversityStudents;
   [1 to 100] Scalar Integer InstanceBind UniversityProfessors;

   Scalar Integer MaxCoursesForFTStudents = 4;

   Scalar Integer MaxCoursesForPTStudents = 2;

   Scalar Integer PassRate = 70;
}
```

Example of a Contracts Parameter Section

In addition a contract may include a section that defines a set of variables. These contract-scope variables are dynamic variables and have the same lifespan as the contract instance to which they belong. The instance variable is an example of a contract-scope variable.

2.6.5 Scenarios

The primary element of an ACL contract in the scenario (pg. 153-163). A contract may contain zero or many scenarios. Scenarios define a sequence of events or responsibilities that when executed in a specified order accomplish a specific task. In effect, scenarios define the temporal aspects of the IUT. A scenario begins with the *trigger* statement. The trigger statement contains the condition (a responsibility or event) required for an instance of the scenario to be initialized (pg. 160). In addition a scenario may contain a terminate statement. The *terminate* statement performs the opposite of the trigger statement in that it defines the responsibility or event that terminates the given scenarios instance. Scenarios may also contain their own local variables, and check conditions for use within the scenario (pg. 162). The following illustrates a simple scenario from the university contract that enforces the requirements to terminate a professor followed by cancelling the courses the professor has been assigned to.

The scenario illustrates a *Trigger* being followed by a grammar of responsibilities and then a *Terminate* statement. In the above example, the Terminate MUST be preceded by an observe statement specifying the event that enables the termination of the scenario. The Trigger statement indicates the scenario is initialized upon the observation of the *TermStarted*.

```
Scenario TerminateProfessor
{
      Trigger(u.TermStarted),
      TimeThread[9],
      {
         observe(TerminateProfessor(tProfessor)),
         DestroyProfessor(tProfessor),

         each Courses()
         {
            choice(iterator.professor) == tProfessor)
            {
                CancelCourse(iterator),
                fire(CourseCancelled(iterator));
            }
         }
         fire (CoursesCancelled(professor));
      } Terminate ([observe(u.TermEnded)]);
}
```

Example of a Simple Scenario

Once the scenario has been initialized the next observe statement indicates that the scenario must wait until the *TerminateProfessor* event has been observed. Here we introduce a parameterized event. Parameterized events are currently not defined in the ACL specifications and as such will be defined further in Chapter Four.

Once the TerminateProfessor event has been observed, the scenario proceeds to execute the *DestroyProfessor* responsibility. Once completed, the *DestroyProfessor* responsibility fires a *ProfessorDestroyed* event (not shown in this example) and control is returned to the scenario. At this point it loops (the *each* statement[10], pg. pp) through the set of courses bound to the contract

[9] The TimeThread is introduced in this thesis and is not defined in the existing ACL specifications

[10] An each statement is used to denote a block of ACL code that is iterated trough for each element of a specified collection. The each statement provides a build-in iterator for variables that are declared using the List variable type

and determines, through the parameter of the *choice* expression[11] (pg. 95), if the course was assigned to the professor. If the choice expression evaluates to *true* it proceeds to execute the *CancelCourse* responsibility. Notice the use of the ACL type *iterator*. An iterator automatically takes on the same type as the type defined in the loop structure.

Upon completion, the CancelCourse responsibility returns control and the scenario fires the *CourseCancelled* event, passing the course that was cancelled as a parameter. The loop continues until all courses have been evaluated and then the scenario proceed to fire the CoursesCancelled event, identify the professor the courses were cancelled for. The scenario then returns the top of the inner structure of the scenario and waits again until another TerminateProfessor event is fired. The scenario continues to be active until the *TermEnded* event have been observed- identifed by the Terminate statement.

2.6.6 Stubs

A stub responsibility is a place holder for one or more responsibilities. Responsibility stubs are analagous to stubs found in use case maps. That is they represent a point where different functionality can be can be either statically or dynamically placed in to the location represented by the stub [52]. In effect, a stub is a place holder for ACL code.

```
stub Responsibility DoMidterm(tCourse c, tStudent s)
{
     Contract Course course = c.bindpoint;
     Check(course.Parameters.MidTerms > 0);

     loop (1..course.Parameters.MidTerms)
           DoMidterm(course,course.MidTerms(iterator)));
}
```

Example of a Responsibility Stub

[11] A choice statement is used to selectively execute a given block of ACL code based on a condition (the expression). A choice statement is similar to the if and switch statements found in other programming languages.

Stub responsibilities must contain a body and are not bound to an IUT method, but rather bound to another responsiblity within the contract (pg. 146). As shown in the above example, the stub is bound to the DoMidterm respoinsibility defined elsewhere in the contract and may contain checks and any other ACL elements that are allowable in an ACL responsibility.

2.6.7 Atomic Statements

An atomic element is used to indicate that all responsibilities and all other contract statements defined by the grammar are to be executed and evaluated as one functional unit (pg. 134). From the context of the scenario, an atomic element is to be considered an atomic unit and may be considered self-contained.

```
Scenario RegisterForCourses
{
      Scalar tCourse course;
      Contract University u = instance;
      Check(u.RegistrationOpened()),

      Trigger(observe(CoursesCreated), IsCreated()),
      choice(IsFullTime())
      {
        (
           atomic
           {
              course = SelectCourse(u.Courses()),
              choice(course.IsFull() | course.IsCancelled())
              {
                 course = SelectCourse(u.Courses()),
                 redo[12];
              };
           };
           u.RegisterStudentForCourse(context, course),
           RegisterCourse(course);
        )[0-u.Parameters.MaxCoursesForFTStudents];
}
```

[12] A choice statement may contain an ACL redo statement. The redo statement indicates that the execution is to be returned to the choice expression and the expression is to be re-evaluated. A redo statement is only valid inside a choice statement and cannot be used outside of this context (pg. 99).

Any variable declared within an atomic unit is local to the atomic statement and is not accessible outside the atomic block. In other words, only those responsibilities, events and the order specified defined within the atomic section are considered valid.

In the above example the atomic statement is considered a single functional unit and must be executed as such before either the *RegisterStudentForCourse* or the *RegisterCourse* responsibilities can be executed. If any part of the atomic unit fails, the entire unit is said to fail. Atomic elements cannot be nested neither can they contain parallel elements.

2.6.8 Parallel Blocks

A parallel block is used to specify that all responsibilities and other statements enclosed with one or more operators. Parallel blocks can be viewed as a single responsibility and may be executed in parallel at any given time (pg. 135).

```
Scenario TakeCourses
{
   Trigger(observe(TermStarted)),
   parallel
   {
      Contract Course course = InstanceBind CurrentCourses(iterator);
      Check(CurrentCourses().Contains(course.bindpoint));

      (
         parallel
         {
             DoAssignment(course,context))[course.Parameters.NumAssignments];
         }
         |
         parallel
         {
            DoMidterm(course,context)[course.Parameters.NumMidterms]
         }
        |
        DoProject(course,context)[course.Parameters.HasProject]
     ) [observe(TermEnded)],
     DoFinal(course,context)[course.Parameters.HasFinal];
   }[CurrentCourses().Length() | DroppedCourse (course)],
   Terminate(Fire(CoursesCompleted, context));
}
```

Parallel block may be viewed as sub-scenarios with a parent or main scenario, and there can be a number of sub-scenarios active at any given time. It is important to note that all sub-scenarios must be completed before an element outside the parallel block can be executed. In the preceding example you have one main outer parallel statement and two inner parallel statements. For the outer parallel statement, the conditional clause at the end of the parallel block indicates that the embedded grammar or ACL code will be executed in parallel for each course the student is registered in (indicated by the *CurrentCourses*().Length() statement), or if the student drops the course the current instance of the outer parallel block is terminated.

The inner parallel statements, can be considered sub-scenarios in that for each course the student is registered in, there can be the *DoAssignment*, *DoMidterm* and *DoProject* scenarios running concurrently (indicated by the ACL ***or*** operator "|"). Only after the inner scenarios have completed, or the TermEnded event has been observed can the DoFinal responsibility be executed.

In effect, you have one main outer scenario, namely *TakeCourses*, one main inner scenario for *each course* the student is registered in, and *three inner scenarios* for each course. An example of a UCM that represents the above parallel block can be found in Figure 3.

2.6.9 Invariants, Checks and Pre and Post Conditions

ACL includes elements derived from the Design-by-Contract paradigm. These are pre and post conditions and invariants. Pre and post conditions are applied to responsibilties and used to check the state of the IUT before and after the execution of the bound procedure identified in the responsibility (pg. 128-129).

```
Invariant SizeCheck
{
   Check (students >= 0);
   Check (students == Size());
}
```

Example of an Invariant Definition

Invariants are also used to check specific states or condition of the IUT, but are checked both before and after the execution of every procedure to which the responsibilities are bound (pg. 110).

In addition ACL defines a check statement. Check statements must evaluate to true or otherwise the check fails. All invariants contain check statements. An oservability may also contain check statements (pg. 112). In the above example the university contract instance has zero or more students and that the number of students the contract has recorded (or created) matches the number of students the IUT class has recorded (or created).

2.6.10 Events

ACL supports events and event monitoring. Events in ACL are occurences that may be relevent outside of the context (or contract) in which they occur. All contract instances are notified of an event occurance whenever an event is fired with the *fire* statement. Only those contracts that contain an *observe* statement for the fired event will take any action when the event if fired. You can see this in the above example for the *DroppedCourse* observability. You will notice we use the *not* keyword to indicate that while the course has *not been dropped*. In effect, while the event *LastDayToDrop* has not been fired, the IUT method *DropCourse* (course) bound to the contract may be executed.

2.6.11 Fire Statement

The fire statement is used to fire the identified *observable event*. Observable events are observed by the observe statement. An observable statement cannot be placed in the body of a responsibility, however they can be placed in an observability or a scenario. A fire statements begins with the fire keyword and is followed by an event identifier. The event identifier does not have be unique within the contract or set of contracts in that multiple fire statements may fire the

same event (pg. 129). Examples of the fire statement are found in the TerminateProfessor scenario example.

2.6.12 Observe Statement

An observe element can be used in the body of the executing grammar. It denotes that execution does not continue until the observable event, identified by the given identifier is observed. Observable events are fired by the ACL fire statement (pg. 142) and cannot be contained in a responsibilities grammar. Examples of the observe statement are found in the TerminateProfessor scenario example.

2.6.13 Bind Points

As discussed in Chapter One, during runtime validation an instance or set of instances of the ACL contracts run alongside an instance of the IUT. More specifically, the VF executes the ACL model, requesting information from the executing IUT whenever needed. Given that the ACL model is implementation-independent, the VF requires additional information, namely a set of bind points, or bindings. Bindings map 1) each class of the IUT to the contract or contracts they must obey, and 2) each responsibility of each contract to a procedure of the IUT.

In effect, bindings bridge the gap between model and implementation. They are the mappings between elements of the contracts and the IUT and eliminate the need for the manual specification of glue code. Glue code requires a programmer to make abstract tests become operational (i.e., usable for validation) via programming [83].

Bindings are crucial for two reasons, 1) allows the contract to be independent of implementation details; i.e. specific type and method names used with the candidate IUT *do not* have to exist within the contracts, and 2) allows several candidate IUTs to be executed against a single TRM (Traceability Requirements Model) and the related ACL contracts. However,

currently ACL has no automatic binding tool and all bindings must be manually identified and coded.

```
Responsibility RegisterCourse(tCourse course)
{
     Pre(CurrentCourses().Contains(course) == false);
     Execute();
     Post(CurrentCourses().Contains(course) == true);
}
```

Example of a Bound Responsibility

In the above example, the Execute statement indicates where the responsibility is bound to the IUT and the sequence of events that must occur before and after the execution of the IUT call.

```
Observability List<tCourse> CurrentCourses();
Observability List<tCourse> CompletedCourses();
```

Example of a Bound Observability

In addition, an observability may be bound to either an attribute or method in the IUT. Although not defined directly, the *CurrentCourses*() and *CompletedCourses*() observabilities would be bound to their corresponding methods or attributes in the IUT. Bindings and their applications will be discussed in detail in Chapter Four.

2.7 Runtime Verification

Verification is the process by which a software system is evaluated with the purpose of providing evidence that the system conforms to the requirements model [83]. Software verification traditionally comprises of methods such as theorem proving, model checking, and testing. However, more recently runtime verification has emerged as a relatively new approach to software verification [84] .

According to Leucker (2012) runtime verification is the application of runtime verification techniques that allows for the dynamic testing of whether or not an implementation under test satisfies or violates the requirements model at runtime. Runtime verification (RT-Verification) is

generally achieved through the use of runtime monitors derived from some high-level specifications [24].

In terms of monitoring an IUT, a run is understood as the possible infinite sequence of system states formed by the current variable assignments or a combination of input/output actions of the IUT. On the other hand, an execution is the finite prefix of a run (a finite set of variables), or more formally a finite trace [84]. With respect to RT-verification we are concerned with reading an execution of an IUT and generating a verdict based on those readings. This is the primary function of a single or set of runtime monitors [24].

2.7.1 Taxonomy of Runtime Monitoring

While a detailed discussion of the taxonomy of runtime monitoring is outside of the scope of our thesis we will provide a brief overview of the aspects of run time monitoring that directly applies to our proposal.

One of the primary concerns and classifications for runtime monitoring is the application area. This will control how the monitor is designed and its primary purpose. Of the four application areas runtime monitoring addresses, the most common is safety checking, followed by security conditions [24]. However, while safety checking is important to our thesis, we are primarily interested in that runtime monitoring may be used to collect information on the IUTs execution in addition to providing performance evaluations.

According to Leucker (2012), runtime verification solutions may use one of three parts of a run: 1) the IUTs input and output behavior, 2) one of its state sequences, or 3) a sequence of events derived from the IUTs execution.

The next category is trace. Trace refers to the type of trace the runtime verification monitor uses to track the execution of the IUT. Regardless of the trace mechanisms, monitoring of a trace will eventually result in some form of validation (true or false) for the property being monitored.

The next consideration is how to monitor the IUT. In other words, how will the monitors actually know or gather the required information necessary to identify what is happening, or what has happened to the IUT during runtime. This is known as integration and may be accomplished in either one of two ways; the first is inline, by interweaving code within the IUT, the second is outline, where the monitoring code can be used to synthesize a monitoring device. Another aspect to consider when discussing monitors is staging. Staging refers to the stage in the IUT execution (or when) verification occurs. Online staging refers to when the monitor is checking the execution at runtime, where offline staging refers to checking a finite set of recorded execution after runtime.

In addition we must consider the fact that monitors may directly, or indirectly influence the IUT during execution in one of two ways. The first is monitor interference. Monitor interference refers to whether or not a monitor impacts the running of the IUT by running on the same hardware. The monitor is said to be invasive if running on the same hardware and non-invasive if running on separate hardware. The second is whether or not the monitor influences or takes action to steer or correct the IUT. As such, monitors may be labelled passive or active.

In addition to the above aspects to runtime monitors, Beltramin [24] identified two additional components we must consider. The first is the target language, or the target language of the IUT that the monitors must address. Target languages may be Java or others.

The second are the categories for specification type. The specification type defines the type of monitoring of the IUT that will be implemented; i.e. Assertion Based, Temporal Assertion Based, State Based or Scenario Based. Assertion-based specifications are those that only check conditions

of the IUT such as pre and post conditions and invariants. Temporal assertion based specifications define expected system operations in terms of properties specified using some form of temporal formulism such as temporal logic or context-free grammar (CFG). State-based specifications specify changes in the state of the IUT. More specifically they focus on a sequence of states where a state is a set of values of some variables at some point in time. Finally we have scenario-based specifications. Scenario-based specifications define a specific sequence of events within the IUT where an event is identified as an occurrence of the invocation of a method or series of methods.

2.8 Summary

In this section we presented three different approaches, i.e. model-based testing, event-driven testing and scenario based testing and how they apply to runtime validation. In addition we discussed their strengths and limitations and their relevance to our thesis. We then introduced use cases and use case maps and their relationship to scenario driven testing. We closed with providing an overview of the monitoring requirements and the main ACL component we address in our thesis with respect to ACL/VF.

3. The ACL Validation Framework

As discussed in Chapter One, ACL has a number of inherent issues, namely its dependence on external tools and the current limitation on the number of contracts and scenarios it can monitor simultaneously; the most prominent of which is the monitoring of multiple scenarios running simultaneously.

3.1 Monitoring Requirements

The key issue is the triggering of scenarios, which controls when an instance of a scenario is initialized. The triggering of a scenario may rely on the observation of a *parameterized event*[13] and the successful *triggering*[14] of a scenario for each distinct instance of the list of parameters.

```
Scalar tCourse course;
Contract University u = instance;

Check(u.RegistrationOpened()),
Trigger(observe(CoursesCreated), IsCreated()),
choice(IsFullTime())
{
   (
      atomic
     {
         course = SelectCourse(u.Courses()),
         choice(course.IsFull() | course.IsCancelled())
         {
            course = SelectCourse(u.Courses()),
            redo;
         };
     };
     u.RegisterStudentForCourse(context, course),
     RegisterCourse(course);
   )[0-u.Parameters.MaxCoursesForFTStudents] | [u.RegistrationClosed()];
}
```

The above ACL code snippet is from the university case study student's contract *RegisterForCourses* scenario.

[13] Currently the ACL specifications does not support parameterized events. As such, we introduce parameterized events in this thesis and recommend they be added to the ACL specifications. We will be discussing this new event type in detail later in this chapter.

[14] A detailed discussion of ACL triggers and parameterized events can be found in Chapter Four.

In the above example you can see that any given instance of a student may register in a course during a specified time frame, i.e. from the registration opened to the registration closed period. In effect the scenario has a life span; there is an identifiable time frame during which a student may register for a course. In addition there may be multiple students registered with the same university performing the same action, or scenario simultaneously. This requires that each instance of the university contract be uniquely identifiable, each student registered with the university be uniquely identifiable, and each course must also be uniquely identifiable. In addition, a contract may have multiple scenarios that must be monitored, each with their own lifespan. As such the lifespan of the scenarios may overlap thus requiring concurrent monitoring.

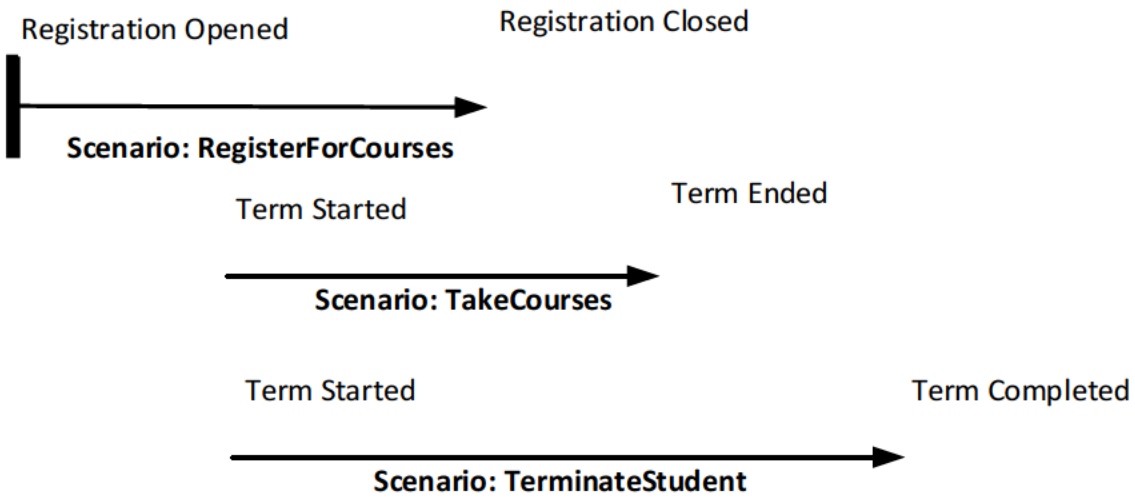

Figure 4: Example of Overlapping Scenarios

In the example above the *RegisterForCourses* and the *TakeCourses* scenarios (identified in section 2.6.8) of the student contract have a period in their lifespans that overlap; the period from when a term starts to the period when registration is closed. In addition the *TerminateStudent* scenario (ACL code below) has a lifespan that covers the period from when a term starts to when

a term is completed. The above figure shows that for any single instance of the student contract, there can be one, two or three scenarios running concurrently, each requiring monitoring.

```
Scenario TerminateStudent
{
   Trigger(observe(u.TermStarted)),
   TimeThread,
   {
      observe(StudentTerminated,tStudent),
      Check(tStudent == context);
      DropCourse(context);
      fire (StudentTerminated(context));
   } [observe(u.TermEnded)];
}
```

This is similar in nature to use case interweaving discussed in section 2.4. This requires that each instance of a scenario within any given instance of a contract must be uniquely identifiable and monitored (concurrently) throughout their lifespans.

3.2 Java Threads

Java threads and the J2EE framework is well suited to address this issue. A thread can be started and stopped by external events, and in addition threads can be grouped together into thread groups. An external event is used to control the lifecycle of threads and control the flow of inter-thread communication, including when a thread can lock a specific scenario or yield and pass control over to another scenario.

In addition thread groups can be monitored and controlled as a single entity. For example a set of student threads can be started and stopped simultaneously based on a single event [82, 83, 84]. Tying scenarios to threads and thread groups allows the VF to control the lifespan of scenarios either individually or grouped together.

Another important aspect to consider is that threads have their own unique identifier assigned by the JVM and cannot be changed or altered during their lifespan. As such, each scenario would

be uniquely identifiable by its associated thread identifier and the hashcode of the enclosing instance of the contract.

The following example shows how we create and assign threads to the contracts thread group and control its lifespan. A scenario is initialized by an external event (e.g. TermStarted). At this point we will note that the Scenario class is an abstract class that all scenarios extend from. This will be discussed in further detail later in this section.

```
protected Scenario (Object o, String name)
{
    super();
    this.name = name;
    ACLContract c = (ACLContract)o;

    setContext(c);
    setContractMonitor(c.getContractMonitor());
    setMonitorStation(c.getMonitorStation());

    _thread = new Thread(this);
    _thread.setName(name);
    _thread.setPriority(Thread.MAX_PRIORITY);
    _thread.start();

    c.addThread()thread);

    BlackBoard.getInstance().addObserver(this);
}
```

Upon initialization of a scenario we obtain a handle to the current contract instance the scenario is bound to. We initialize a thread and bind the scenario to it. We then provide a unique name, set the priority (scenario threads are assigned the highest priority) and start it. We then assign the thread to the contracts thread group that was created when we initialized the enclosing contract. You will also notice that the scenario is added as an observer for the blackboard. More on this later in this section.

3.3 Contract Monitors and Monitor Stations

In this section we will discuss in detail the contract monitors and monitor stations we propose for our solution. Since our solution is based on previous research by Beltramin [24] and his work

with JavaMOP we will provide a brief overview of the relevant aspects of JavaMOP before we go into detail of our solution. For our discussion we will present the ReportMarks scenario found in university test case course contract.

```
Scenario ReportMarks
{
   Trigger(observe(TermEnded)),
   TimeThread,
   each(Students())
   {
      u.ReportMark(context, iterator, MarkForStudent(iterator));
   },
   Terminate(fire(MarksRecorded,u));
}
```

ACL Code for the ReportMarks Scenario in the Course Contract

3.3.1 Review of JavaMOP Monitors

Previous research conducted on mapping ACL specifications to the JavaMOP framework [24], more specifically scenarios and atomic elements showed that these mappings can be supported by the JavaMOP framework. However, no research for the support parallel blocks was conducted at the time.

Each ACL contract specification has a corresponding monitor (or set of monitors) in JavaMOP. These monitors include variables that can be used as dynamic test cases for runtime checks as well as support logical formalization of context-free grammars to verify the temporal grammar and execution of events. In addition a sub-system to JavaMOP was added that supported the firing and observation of global events, that is, events that are visible to all contracts. The following is an example of the JavaMOP code required for the *ReportMarks* scenario in the course contract.

Any JavaMOP monitor may fire an event to the Blackboard, or observe the Blackboard for a specific occurrence (firing) of an event. The Blackboard is also responsible for managing contract class instances. These class instances corresponded to contract instances and contain elements such

as contract variables, observabilities and invariants. As such all monitors related to a single instance of a contract as well as the contract scenario instances are registered with the blackboard.

```
Course_Scenario_ReportMarks (Course course)
{
   Course context;
   Course_Contract contract;
   University_Contract uContract;
   int numMarksRecorded = 0;

   creation event Trigger after(String id, Course course, University uni):
      execution(* Blackboard.Fire(..))
             && args(id, course, uni)
             && if (id.equals("TermEnded_Trigger"))
      {
      uContract = Blackboard.UniversityInstances.get(uni.toString());
      contract = Blackboard.CourseInstances.get(course.toString());
      context = course;
   }

   event ReportMark after(Integer mark, Course course, Student student):
   execution (* University.ReportMark(..))
          &&    args(course, student, mark)
          &&    condition(mark == contract.MarkForStudent(student))
   {
      numMarksRecorded++;
      Blackboard.Fire("Terminate_ReportMarks", context);
   }

   event Terminate after(String id, Course course):
   execution(* Blackboard.Fire(..))
          && args(id, course)
             && condition(id.equals("Terminate_ReportMarks")
             && (numMarksRecorded == contract.Students().size()))
      {
      Blackboard.Fire(new ACLEvent("MarksRecorded", context));
   }

   cfg : S -> Trigger ReportMark* Terminate

   @fail {
         throw new ScenarioGrammarViolationException
                              ("Course ReportMarks Scenario Grammar Violation");
   }

   @match {
         System.out.println("Course ReportMarks scenario grammar match. Monitor
                               reset.");
      __RESET;
   }
}
```

Example of JavaMOP Source Code for the Report Marks Scenario in the Course Contract

In this manner the Blackboard keeps track of contract classes through a mapping from an IUT instance ID to its corresponding contract class instance. Monitors can then use their local reference to the instance they are bound to access the instance of their corresponding contract class.

In JavaMOP a header contains the generic information required to control the instantiation of a contract or scenario instance and the binding of the scenario instance to the contract instance. It consists of a set of modifiers, ID and parameters. The ID is the unique identifier for the monitor. The parameters in a monitor header denote the set of parameters that the current monitor is bound to (i.e. the instance of the university contract, the course contract and the actual IUT course instance). This allows for the unique generation of a monitor based on its parameters. For example, in our previous example, a unique monitor would be generated for each pair of student/course pair identified for an instance of a university.

Recall from Chapter One, we mentioned three issues with the JavaMOP solution, the first being the complexity of the code and code generation. If you notice the simple nature of the ACL code for the scenario and the complexity of the corresponding JavaMOP code. The JavaMOP code has to be written from scratch for each contract and each scenario. And as you will notice it requires specialized skills that focus on AspectJ and JavaMOP.

3.3.2 Proposed Solution

In our solution we introduce a new element, the monitor-station. During runtime validation each instance of the IUT has a corresponding instance of a monitor-station and both the instance of the IUT and monitor station run in parallel and communicate between each other during RT-validation.

3.3.2.1 The Monitor Station

The monitor-station is responsible for initializing and monitoring the instances of each ACL contract. In addition the monitor station is used to provide a central location for global scoped variables and events so they can easily be accessed across contract monitors.

In our solution instead of passing through a set of parameters to the contract monitor we bind the instance of the IUT object directly to the contract. For example, in the university case study the university contract is the main contract. For each instance of the university contract initialized during runtime validation, a corresponding monitor station is initialized. The instance of the IUT university object is bound to the monitor station.

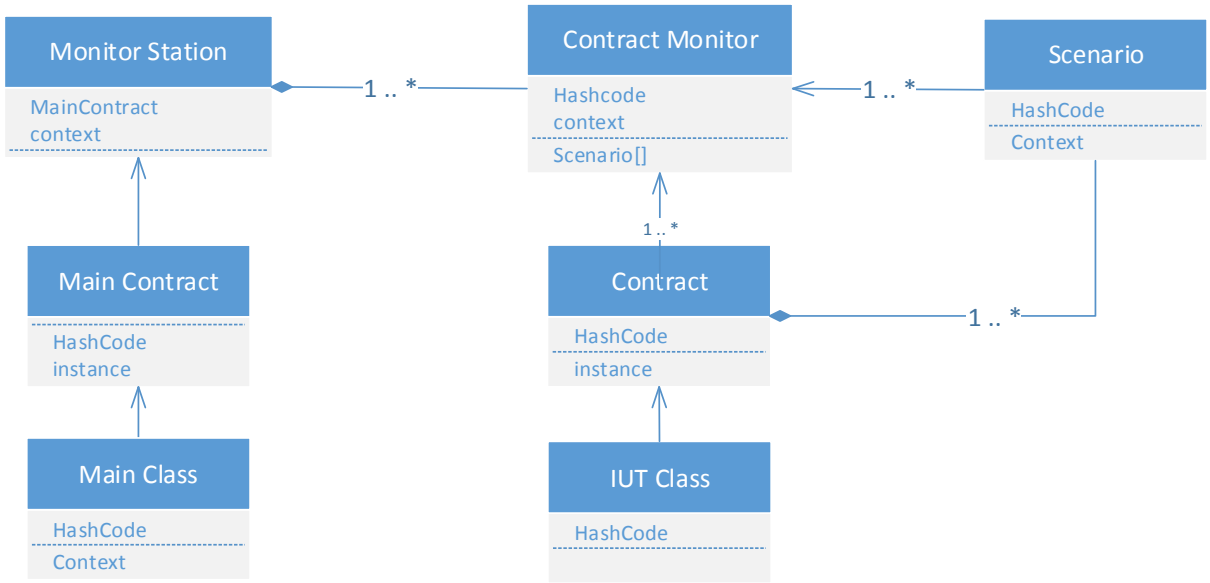

Figure 5: Relationship between ACL/VF Monitors and ACL Contracts

Recall that a university may have multiple students and multiple courses associated with it. Each student object that is initialized by the IUT is bound to the student contract monitor. At this point, a thread (a contract thread) and a contract thread group is created for each student contract initialized. In addition, for each scenario defined in the student contract, a corresponding thread is initialized and assigned to the contract thread group. Unlike the JavaMOP solution where there

was a single monitor created for each scenario, the contract-monitor is now responsible for managing the lifespan of the student contracts. In effect, contract monitoring is now located in a single instance of a contract-monitor. The same process is carried out for each of the course contracts initialized during the RT-verification process.

Through reflection we create instances of the IUT objects and bind them directly to the contract. Reflection is a feature in the Java programming language that allows an executing Java program to "introspect" itself and identify and manipulate internal properties of the program. For example it is possible for a Java class to reflect upon itself and obtain its class name, constructors, methods and attributes, we can even determine the methods parameters and return types.

Through reflection the IUT parameter is bound to associated contract scope variable during contract initialization. Each contract-monitor during initialization is assigned a hashcode, and each corresponding IUT object is bound directly to the contract through by the context variable, in effect allowing direct access to the hashcode of the bound IUT object. This ensures that the IUT is uniquely indefinable within a contract instance. Recall that during the initialization process a thread is created for each contract instance. This is handled by the *MonitorHelper* class when we create the instance of the contract-monitor. Once initialized the contract monitor is registered with the Blackboard. More on the Blackboard later in this chapter.

3.3.2.2 Contract Monitors

Contract monitors are used to provide a central location for global scope variables so they can easily be accessed across all contracts bound to the contract monitor. In addition contract monitors are responsible for managing the lifecycle of the contract.

Each contract monitor has a corresponding contract class that includes the contract variables, observabilities, invariant checks and the scenarios. As an IUT instance is created at runtime, the

contract class corresponding to the contract the instance is bound to is initialized and the contract class is then bound to the contract monitor. The observabilities, responsibilities, scenarios and invariants are defined within in the contract class itself, and all events are monitored by the contract monitor and pushed to the contained contract. When the contract fires an event it fires it directly to the containing contract monitor and the contract monitor is responsible for managing the lifecycle of the event.

Contract monitors also provide a mechanism that makes global scoped variables accessible across all instances of the contracts bound to it. For example, in the university case study the university contract has a number of variables that are applicable across all student contracts.

```
Parameters
{
   Scalar Integer MaxCoursesForFTStudents = 4;
   Scalar Integer MaxCoursesForPTStudents = 2;
   Scalar Integer PassRate = 70;
}
```

These parameters are global in scope and are bound to the contract monitors when they are initialized. It is important to note that these global variables are static across all instances and cannot be modified.

The following is sample code from the monitor station that is responsible for creating instances of the contract class and contract monitor class and binding them to the monitor station. We will walk through the sample code and explain how the contract monitors and contract classes are generated and discuss the binding process.

A contract class is initialized and an instance of the IUT class is passed in as a parameter. The *MonitorHelper* is a helper class that uses reflection on the parameter passed in to create an instance of the contract class and bind the IUT object to the contract object. A contract monitor is then instantiated and the instance of the ACL contract is passed through as a parameter. The contract is

then bound to the associated contract-monitor. Every time a contract or contract-monitor is instantiated the Java Virtual Machine (JVM) assigns a unique hashcode to it.

```
public void register (Object o)
{
   UniversityContract uContract =
         (UniversityContract) ContractHelper.createContract(o);
   UniversityContractMonitor uMonitor =
         (UniversityContractMonitor) MonitorHelper.createMonitor(uContract);

   uMonitor.setMonitorStation (this);
   uContract.setMonitorStation (this);

   setContext (uContract);
   setContractMonitor (uMonitor);

   BlackBoard.getInstance().addObserver(this);
   ReportWriter.getInstance().registerMonitor(uMonitor);
}
```

The contract-monitor and the contract are bound together through their unique hashcodes. The contract-monitor is registered with the monitor station and the monitor station is registered with the Blackboard and ReportWriter (Contract Evaluation Report Engine). The contract-monitor is now responsible for starting up and managing the lifespan of the contract.

```
@Override
public void run()
{
   TimeThread tmTimeThread =
                          ((UniversityContract) getContext()).new Term(this);
   StartThread (tmTimeThread,"TimeThread[University.Term]");
}
```

In the example above the contract monitor starts the university contract's "Term" scenario and passes an instance of the contract-monitor to the scenario. Once a scenario has been identified it is passed through to the contract-monitor that controls the instantiation of the scenarios. Recall that a thread is assigned to each scenario and the thread is then assigned to the contract-monitors thread group. You will also notice that we pass through a scenario name. This is an optional parameter that can be used to further refine the information presented in the reports generated during runtime verification.

```
protected void startThread (TimeThread tThread, String name)
{
   Thread t = new Thread (tThread);
   t.setPriority (Thread.NORM_PRIORITY);
   t.setName (name);
   t.start ();

   threadGroup.add (t);
}
```

3.3.2.3 The Station Manager

The Station-Manager is the main class for the ACL/VF and has two primary functions, starting and stopping the RT-Validation process, and initializing and binding the IUT to the monitor-stations.

As mentioned above, during runtime validation each instance of the IUT is bound to a monitor-station and each contract in the ACL project is bound to a contract-monitor. The monitor-station is responsible for managing the lifespan of the contracts bound to it and managing inter-contract communication via the blackboard. The contract-monitor is responsible for managing the lifespan of the scenarios associated with each instance of a contract.

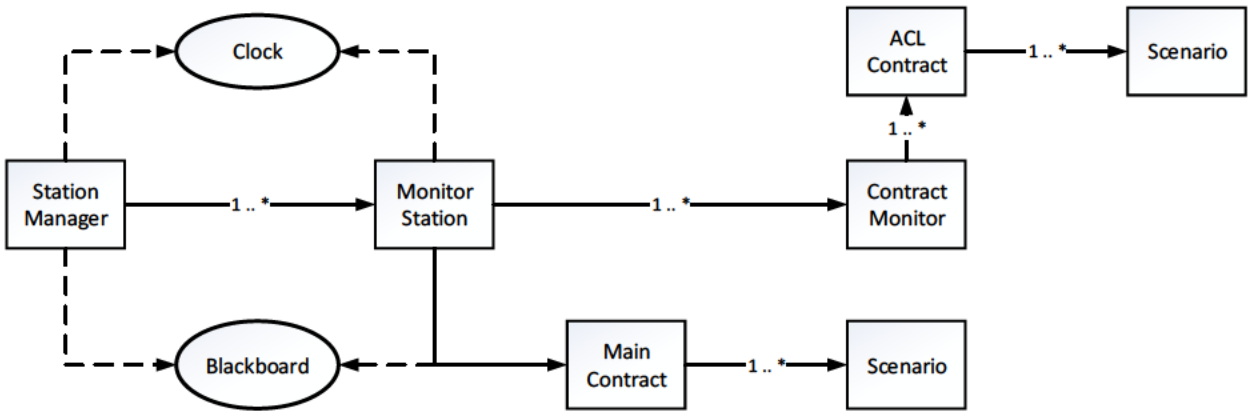

Figure 6: Validation Frameworks Component Diagram

The station-manager is configured through the ACL/VF configuration file. The first parameter identifies the type of monitoring we are performing i.e., active or passive. Active indicates it is running on the local machine, passive indicates it is running on a remote machine.

```
#Monitor Configuration
```

```
#monitor type [passive | default active]
monitorType=active
#IF set to 0 [default 0] 1:1 ratio between Monitors and Contracts
numOfMonitors=0
numOfUniversities=2
```

The second and third parameters indicate the number of instances of the IUT to initialize during RT-validation and the binding requirements for the IUT to their related monitor-stations. It is important to note that we can actually bind more than one instance of the IUT to a single monitor-station. However, as noted the default is to instantiate and bind a single instance of the IUT to a single instance of a monitor-station. In the above example, we create two instances of the IUT and a single instance of a monitor-station for each instance of the IUT (tow in total). Two other parameters, start-time and end-time will be discussed later in the section on the ACL clock.

3.3.2.4 Contract Classes

Contract classes are used to provide a central location for contract-scope variables so they can easily be accessed by all contracts bound to the contract monitor. In addition contract classes provide a central location for managing the lifecycle of the scenarios, atomic statements and parallel blocs defined in the contract. Each contract class has a corresponding instance class bound to it and includes the contract variables, observabilities, invariant checks and the scenarios.

As an IUT instance is created at runtime, the contract class corresponding to the contract the instance is bound to is initialized and the contract class is then bound to the contract monitor. The observabilities, responsibilities, scenarios and invariants are define within in the contract class itself. All events are monitored by the contract monitor and pushed to the bound contracts. When the contract fires an event it fires it directly to the containing contract monitor and the contract monitor is responsible for managing the lifecycle of the event.

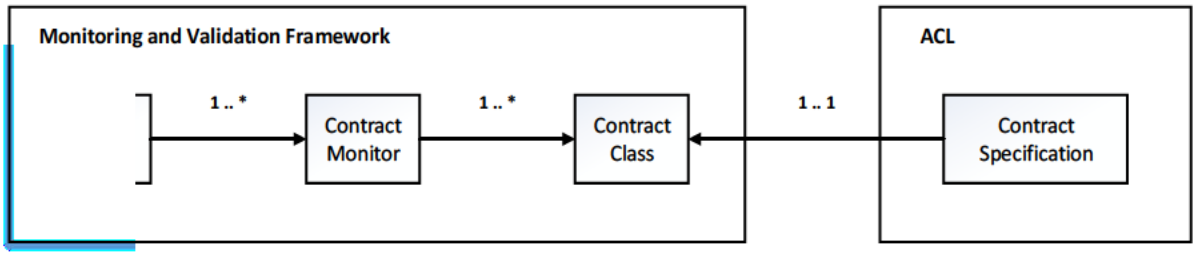

Figure 7: Validation Framework Component Diagram

Up to now we have defined the relation and interaction between monitor stations, contract-monitors and the IUT contracts and illustrated how these relationships are uniquely identifiable by a compound key, namely the hashcode of the monitor station class, contract monitor class and the IUT contracts. We have also demonstrated how our solution simplifies the development of the monitoring code over the JavaMOP solution. Since there was no source code generator at the time of writing this thesis the monitor-station and contract-monitor code was written from scratch.

3.4 The Blackboard

In order for multiple ACL contracts to monitor events that are fired externally, i.e. communication between inter-contracts we need an independent class that all contracts interact with simultaneously. To address this issue we build on the previous work by Joshua Beltramin [24] and expand upon the ACL Blackboard.

The initialization and termination of a scenarios are triggered by observable events. These events may be fired by a responsibility, for example when a specified set of functions has been completed via their associated contract responsibility (i.e. create and register a collections of professors associated with a university), fired by a scenario (i.e. when a scenario is triggered or terminated), or may be time sensitive, for example when a specific time element associated with contract (i.e. a term associated with a university) occurs, or a combination of all three. In order to centralize the location for event management and observation Beltramin introduced the ACL

Blackboard. In effect, the Blackboard is central to inter-contract communication and acts as a central system in which any ACL event can be fired and observed.

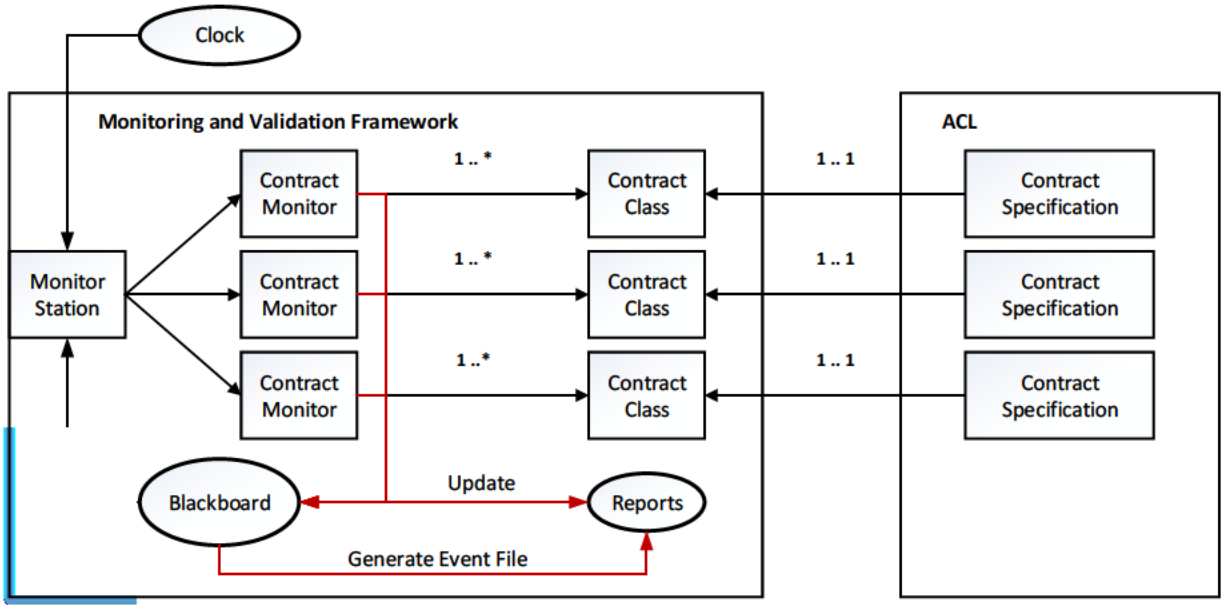

Figure 8: Relation between the main ACL Validation Framework components

The Blackboard is a set of classes that are included in the ACL validation framework. At its core the Blackboard is responsible for handling inter-contract communication and monitoring the firing of events. The Blackboard consists of two classes, the Blackboard itself and the ACL Event. The Blackboard is a singleton class that all monitor-stations are required to implement. The Blackboard is a singleton class that each instance of a monitor-station shares. The Blackboard encapsulates the methods and data structures necessary for the firing and observation of events and tracking the contracts responsible for firing events, along with the monitor station the contracts are bound to.

In the original Blackboard presented by Beltramin [24] the blackboard was responsible for keeping track of the mappings between contract classes and their respective IUT instance. However, in our solution the responsibility keeping track of these mappings has been pushed to

the monitor station. The Blackboard now has a single responsibility – the management of events and inter-contract communication.

The Blackboard functions through two central methods, fire and observe. The fire statement, similar to the original design takes either an ACL event as a parameter or a message string that is then converted to the corresponding ACL event type. Upon receiving[15] an event, the Blackboard pushes the ACL event to the contract-monitors bound to the monitor station[16]. These calls may take additional parameters that further refines the information for the ACL event fired.

However, unlike the original design, the Blackboard is now responsible for keeping track of the events fired and their associated contract monitor. Each monitor station, when initialized is registered with the Blackboard and only a single instance of the monitor station can be registered (i.e. it must have a unique id). However multiple independent instances of a monitor-station (each with their own unique id) may be registered with the Blackboard. Remember from our previous discussion that a monitor-station has a handle to the contract monitor or set of contract monitors, and each contract monitor is associated to an instance of the IUT. All registered monitor stations may poll the Blackboard and via their unique key determine if a specific ACLEvent has been fired, the time it was fired and which contract and scenario was responsible for firing it. We will discuss it further detail later in the section on ACL events.

3.5 The ACL Clock

A secondary issue arose during our research. In addition to being event driven a scenario may be time sensitive, i.e. the lifespan of a scenario may be reliant upon time segments, start time and

[15] The blackboard is a singleton class and each monitor station has a handle to its instance. When a monitor station fires an event to the blackboard it is now a direct call to the fire method in the Blackboard. The blackboard no longer "listens" or observes events

[16] This is now based on the observer pattern. Each contract-monitor, when instantiated is registered as an observer with the blackboard. When the blackboard receives an event, it notifies the contract-monitors there has been a new event fired. This will be explained further when we discuss the ACL clock.

end time. To address this issue we introduced the notion of an ACL Clock. The clock has two parameters, start-time and end-time which are configurable in the ACL/VF configuration file. In the following example the clock has a start-time set to Dec 30, 2014 and end-time of June 30, 2016. The clock is a static class and only one instance of the clock can be invoked.

```
#Clock Start and Stop Date/Time
#format dd-mm-yyyy

startdate=30-12-2014 00:00:00.000
enddate=30-06-2016 23:59:59.999
```

The ACL Clock is a thread based class that runs until the end-time is reached or is interrupted by the station-manager. There is a mechanism built into the station-manager that allows for the manual interruption, i.e. pause or shutdown of the station-manager. When the clock reaches the end-time, it fires a *Terminate* event to the station-manager and the station-manager shuts down all monitor-stations and cleans up the threads and generates a set of reports.

The clock has additional parameters in the ACL configuration file that allows for the fine tuning of the running of the clock. In the following example the clock is configured to increase the time-stamp by twenty four hours every 1000 milliseconds.

```
#Clock in milli-seconds
waittime=1000
logClockTime=false

#clock increment | in hours
timeincrement=24
```

The Blackboard, the station-manager and monitor stations are registered with the clock. When the time-stamp is incremented by the configured amount, in this case one day, the Blackboard, station-manager and registered monitor stations are notified of the change in time. The monitor stations use this time-stamp to determine if any time sensitive events should be fired, e.g. TermStarted, TermEnded.

The inclusion of the ACL clock allows the validation framework to monitor the temporal grammar of time sensitive scenarios and control the initialization and termination of these related scenarios. In addition scenarios may be cyclic in nature (repeatable) and their termination may be dependent on an external event, in this case a time-stamp (end time). A time-stamp is a specialized ACL event that allows the validation framework to determine if the ACL events were fired and observed in agreement with their associated time segment, i.e. whether the temporal grammar is maintained.

For example, there may be a sequence of events that have intermittent time periods between them, and in addition they may be cyclic in nature (repeatable).

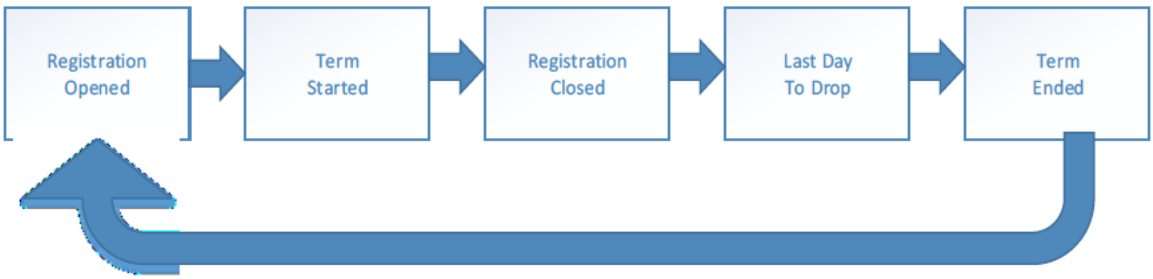

Figure 9: Sample of temporal grammar of a scenario

In the above diagram we show a segment of the temporal grammar of scenario we discuss in Chapter Four. Each event is separated by a specified period of time that is unique to the related IUT. Once the Term has ended the scenario checks the termination condition, i.e. has the ACL clock reached the end-date, if not it repeats the scenario, i.e. it repeats the sequence of events based on the information set for the next term.

3.6 ACL Events

As previously mentioned, ACL events are occurrences that may be relevant outside the contract in which they occur. When an event is fired only those contracts that contain an observe

statement for the fired event will take any action. Events are also be monitored by scenarios, either by the ACL *Trigger* statement or directly through the use of the ACL *Observe* statement. The lifetime and interaction of events is illustrated in the following *event dependent* scenario. Please note, we introduce a new ACL type, parameterized events.

```
Scenario TerminateProfessor
{
   Trigger (Observe(TermStarted)),
   TimeThread
   {
     Observe(TerminateProfessor(tProfessor)),
      DestroyProfessor(tProfessor),

     each Courses()
      {
         Choice (iterator.professor ) == tProfessor)
        {
            CancelCourse (iterator),
            fire(CourseCancelled(iterator));
        }
     }
     fire (CoursesCancelled( professor ));
   } [Observe(TermEnded)];
}
```

First, we provide a detailed walk through of events and discuss how they control the interaction and temporal elements of scenarios. We will then provide a detailed description of events themselves and further define the requirements of parameterized events.

3.6.1 Walkthrough of Event Sequencing and Scenario Management

As you will notice, the scenario is time sensitive and is initialized when a term is started, *TermStarted* event (i.e., start date) and ends when the term is completed, *TermEnded* event (i.e., end date). According to the ACL specifications a scenario is initialized upon the successful completion of a trigger statement. In this example an observe statement is passed as a parameter to the trigger element and upon successful observation of a *TermStarted* event initialization of the scenario is triggered.

Next the scenario implements another observe statement and waits until it is notified that a professor has been terminated, the *TerminateProfessor* event. You will notice this is a parameterized event.

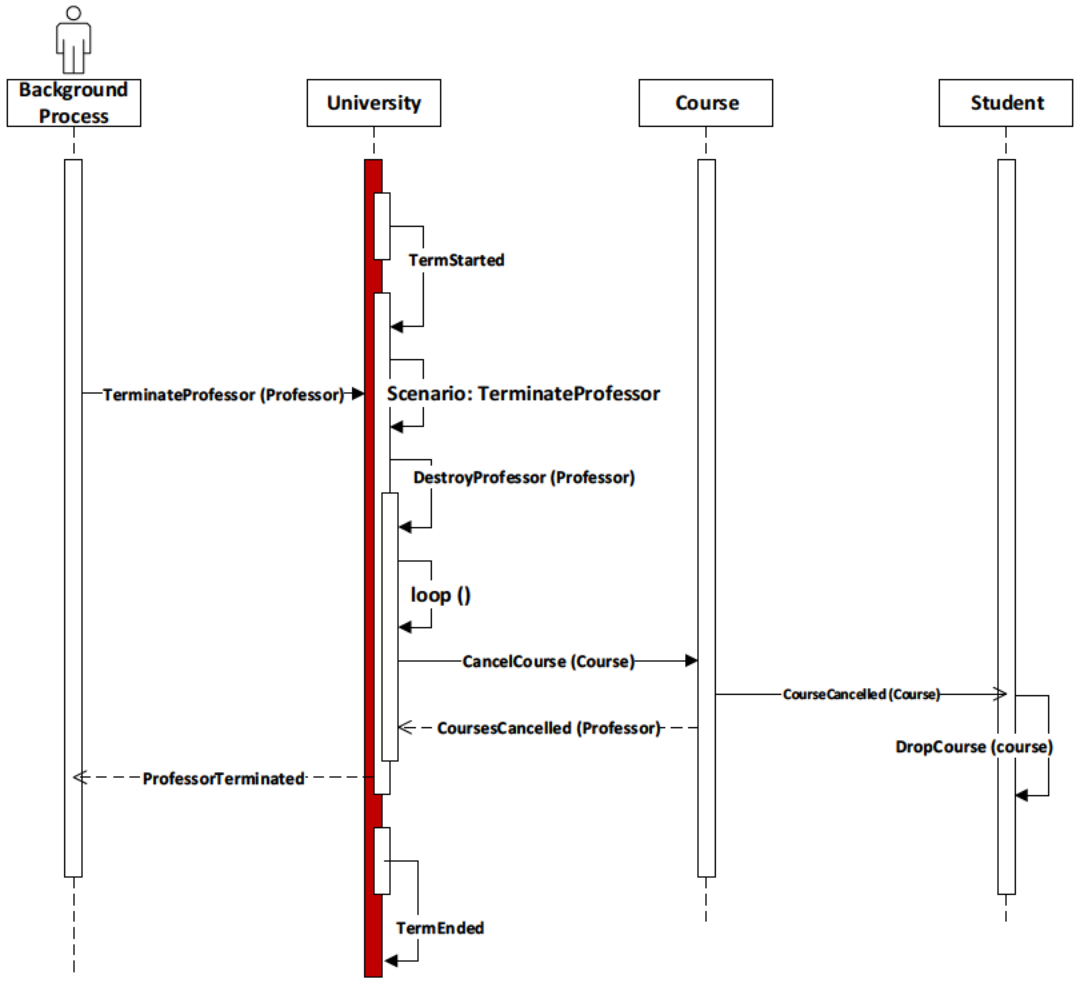

Figure 10: Simplified Message Sequence Chart for the Terminate Professor Scenario

Currently parameterized events are not supported by the ACL/VF. The *TerminateProfessor* has additional information added to it that identifies the professor to be terminated (tProfessor). Once the *TerminateProfessor* event has been triggered the scenario invokes the *DestroyProfessor* responsibility, waits for the responsibility to complete and proceeds to remove the professor from the courses they were assigned to. The DestroyProfessor responsibility fires a *ProfessorDestroyed*

event to the Blackboard passing the terminated professor as a parameter before returning control to the scenario.

The loop structure, identified by the *each* keyword invokes the *Courses* observability and loops through the list of courses associated with the inter-related university contract. The *choice* keyword indicates the action to take if the professor assigned to the course is the same as the terminated professor. If true, the *CancelCourse* responsibility is invoked and the course identified by the iterator is passed as the parameter. Once the responsibility has completed and the course has been cancelled a *CourseCancelled* event is fired. The student contract has a listener (or observable event) that is dedicated to listening for a *CourseCancelled* event. This is covered in further detail later in this section when we discuss observabilities. Upon observing the *CourseCancelled* event the student contract invokes the DropCourse responsibility.

The sequence of events continues until all courses associated with university have been tested. Once the loop is completed a second ACL event, a parameterized event *CoursesCancelled* is fired indicating that all courses associated with the specified professor have been cancelled. All events are logged to the CER.

Once the *CoursesCancelled* event has been fired the scenario checks to see if the term has ended. If not, the time thread returns to the start of the scenario contained within the TimeThread statement and triggers the second observation again. The sequence of events continues until the term has ended.

3.6.2 Enhanced ACL Events

Although not documented in the ACL specifications ACL events are a critical component of the ACL/VF. Originally, events in ACL consisted solely of an ID, as such there was no way to transfer additional information with an ACL event between instances of contracts. In effect, there

was no way to specify the context of the event (i.e., the contract from which the event was fired) or any additional data relevant to the event.

In his thesis Beltramin [24] introduced an enhanced ACL event. The enhanced ACL event not only contained an ID, but also contained a context and data object. These additional attributes were generic objects types and as such was able to carry any type of information. In general, the context attribute carried an instance to the reference to the IUT object the monitor that generated the event was bound to.

Although the enhanced events allowed for greater flexibility for event monitoring between contracts, it was limited with respect to traceability. In addition, since we have introduced time sensitive scenarios we need a way to determine the date and time the event was fired an the monitor-station the contract instance is bound to. Our solution builds on Beltramin's enhanced ACL events and introduces a new set of attributes.

We must first note that the ACL event is now an abstract data type. We now have concrete event types (i.e., TermStarted, TermEnded) that inherit from the abstract ACL event. These event types are user defined and as such allows for the monitoring of specific ACL events. In addition any additional information may be identified and assigned when the ACL event is first initialized, for example, the *TermStarted* and *TermEnded* may have the current term assigned to its data object, the *TerminateProfessor* would have the Professor object assigned to its data object, etc.

To manage this new functionality we introduced an *EventHandler* to the ACL/VF. A user now has the ability to assign user defined events to the *EventHandler*. The *EventHandler* is a singleton class that is incorporated in the VF. All elements of the VF that fire events (i.e., monitor-station, contract-monitors, contract and the clock) implement the *EventHandler*.

In our solution, when an ACL event is originally fired it is 1) sent to the *EventHandler* (along with the instance of the contract or monitor that fired it and an instance of the defined ACL event type is instantiated and is 2) returned to the ACL element that requested it. During the instantiation process two date stamps are assigned to the event, the first is the actual time, or system time the event was fired, and the second is the current date identified by the ACL clock. This allows us to determine the sequence of events with respect to real time and RT-validation. In addition, the *EventHandler* assigns the monitor id, contract monitor id and instance id of the IUT to the event.

Once an event has been returned it is 3) immediately registered with the Blackboard and CER and 4) a corresponding xml file is generated. When an event is 5) observed by another ACL element the event is updated and flagged as observed. The event is 6) updated and the monitor id, contract id and instance id that observed the event is registered with the event, along with the scenario id (thread id). The 7) corresponding xml file is updated. Upon completion of a scenario or group of scenarios the list of events are sent to the CER and a report is generated.

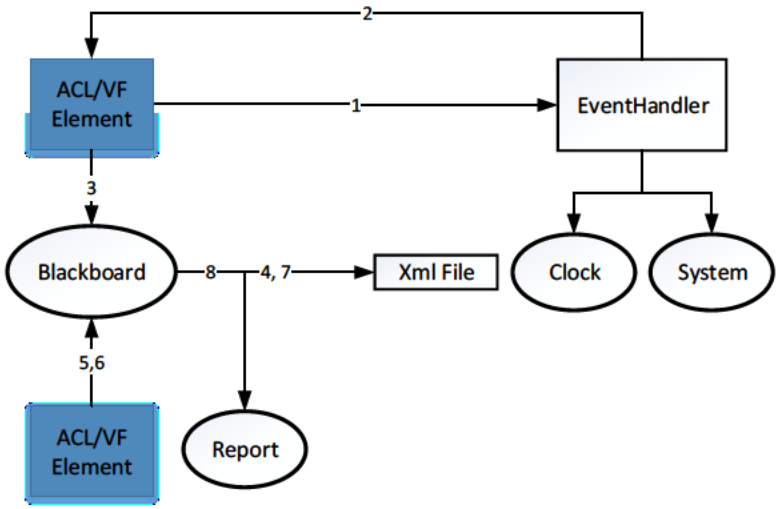

Figure 11: An ACL Event Lifecycle

The refinement to the ACL event structure allows us to drill down further to the scenario level and monitor the temporal aspects of the IUT. We would like to note that currently ACL events are

stored in associated XML files with the systems timestamp of when they were generated as their ID number.

The following is an example of a failed event. A failed event is defined as 1) an event that was not observed, 2) observed, but not in the scope of the contract, or 3) observed out of sequence of the grammar it was defined in.

Examining the xml tags we find the event is traceable back to the instance and class associated with the IUT and the current scenario being monitored. Both the monitor station and contract-monitor and the status of the event (in this case failed) are logged. In this example the event type is the *ScenarioCompleted* event fired by the *ReportMarks* scenario in the course contract.

Further examination shows that the *ReportMarks* event was specifically associated with an instance of the Comp 3203 course. The instance id (hashcode) of the course contract itself and the id of the instance of the course object associated with the IUT are also indicated. In summary the xml file indicates the current status of the event and IUT.

```
<ACLEvent hashCode="256331029" eventId="1438033664071"
          eventName="ScenarioCompleted"
          uid="1465724122" mid="1136497418" term="0"
   <Time>2015-07-27T17:47:44.071-04:00</Time>
   <Observed>false</Observed>
   <Observable>true</Observable>
   <Fired>true</Fired>
   <FiredBy>CourseContract</FiredBy>
   <BaseMonitor>1429880200</BaseMonitor>
   <BaseMonitorClass>UniversityContractMonitor</BaseMonitorClass>
   <MonitorStation>1429880200</MonitorStation>
   <MonitorStationClass>MonitorStation</MonitorStationClass>
   <ContractId>1103172529</ContractId>
   <ContractMonitor>1771604075</ContractMonitor>
   <ContractMonitorClass>CourseContractMonitor</ContractMonitorClass>
   <ThreadId>63</ThreadId>
   <ThreadName>Scenario[Course.COMP 3203.ReportMarks]</ThreadName>
   <instanceid>145286613</instanceid>
   <Data>Scenario[Course.COMP 3203.ReportMarks]</Data>
</ACLEvent>
```

Recall from our discussion on event driven testing in section 2.2 and the six criterions that had to be met. Namely, 1) event coverage criterion, 2) event-statement coverage criterion, 3) event-decision coverage criterion, 4) event-condition criterion, 5) event-decision/condition coverage criterion and 6) event-condition combined coverage criterion. By generating the xml files and the corresponding event reports we are able to determine if we meet the requirements for each criterion. Assuring that we meet the requirements of sufficient event testing.

Up to now we have discussed in detail ACL contracts and their relationship to the contract-monitor and monitor station and the basics of ACL events and how ACL events are managed. We will now turn our attention to how we create a test suite.

3.7 Runtime Validation

In order to be able to manage and control the initialization of multiple instances of the IUT we introduced the concept of a Station Manager (SM). The SM follows the singleton pattern: and only one instance of the SM can be instantiated. As previously discussed the SM instantiates one or more instances of the IUT and Monitor Stations (MS) and binds the instances of the IUT to their assigned MS.

For testing purposes we developed a configuration file that allows us to define the number of instances of the IUT and the set of associated contracts to test concurrently. Each instance of the IUT and its set of contracts with have different values for their attributes, e.g. number of students, number of courses, passing grade etc., in effect creating variations for the same scenario or set of scenarios defined in the contracts. Recall that an instance of the IUT may have multiple instances of associated contracts, so multiple instances of the IUT will have multiple instances of the set of contracts and associated scenarios that are defined in the contracts.

The Station Manager contains a collection of monitor stations that are initialized at runtime and managed throughout the lifespan of the ACL test case. Once the monitor station are initialized we then register an instance or set of instances of the IUT with the monitor station. Recall that the number of monitor stations and the number of instances of the IUT to register with the monitor station are configured in the configuration file.

The following section of code details the logic we follow in setting up and configuring the ACL Validation Framework. First we need to initialize the newly defined ACL Clock. We then proceed to validate that all our required parameters are set as defined in the configuration file through the trace files, the xml test cases generated and the start and stop times of the ACL clock.

```
public void startup()
{
   startClock();
   checkSetupParameters();
   configureStationManager();
   setupMonitorStations();

   try
   {
      Thread t = new Thread (_instance);
      t.start();

      // We need to wait a while and allow the Clock and Blackboard to
      // synchronize before starting up the background processes and
      // monitor stations

      Thread.sleep(ConfigManager.getInstance().waitTime());
                startupMonitors();
      } catch (InterruptedException e)
      {
      // perform a clean shutdown and log the shutdown process
         shutdown(true);
      }
      bp.start();
}
```

We then configure the station manager and instantiate the number of IUT instances to test. Each instance of the IUT is registered with the Station Manager. We then configure and setup the monitor stations; in our example one monitor station per instance of the IUT. A background thread is started, starts up the monitor stations and waits for the termination condition to be met. In this

case we set a start date and end date that is monitored by the ACL Clock. Once the termination condition is met (the end-date) the station manager shutdowns all instances of the monitor stations, contract monitors and cleans up the instances of the IUT. The Station Manager is then shut down and generates a set of reports (See Appendices for examples of the reports).

Another important aspect of the Station Manager to mention is that it is responsible for starting and stopping a set of background processes that are used by the validation framework; for example the ACL clock and instances of test cases that determine when and which professor and/or student to terminate. The background processes have the same lifespan as the station manager, i.e. the lifespan of the ACL validation process.

3.8 Summary

In this section we have addressed five critical issues and provided solutions for each. The first is the issue of the development environment and the identification of an industry language that can support the ACL Validation Frameworks requirements without the need for external or third party tools. This is a critical issue if we want to provide an environment for future research and development and provide scalability. As we have shown the core functionality of the ACL/VF can be implemented in a pure Java environment. This provides full scalable solution independent of any third party tools. Since Java is a development language that is both operating system and platform independent, this allows the ACL/VF to be tested and validated in a variety of operating systems and platforms.

The second issue addressed is the unique identification of events and the monitoring of events. In this section we have shown how, through the use of *hashcodes* combined together allows the framework to identify the firing and observation of these events and maps them directly to the related monitor stations, contract monitors, contract instances, IUT instances and scenarios.

The third issue is the initialization and monitoring multiple instance of the same scenario taking different paths. The solution we put forth is built on the first two recommendations above. By the use of monitor-stations, contract-monitors and Java threads we are able to initialize separate instances of the same scenario and map them directly to the ACL contract via the unique id code mentioned above. Through the use of thread groups the contract monitor and the contract instances bound to the contract monitor are able to group the scenarios together, form a relation between the scenarios, and manage the lifespan of each scenario or group of scenarios.

The fourth issue is inter-contract communication. Inter-contract communication is managed by the Blackboard through the handling of ACL events. We introduced an extended ACL event that provides tracking of the monitor station, contract monitor, contract and instance of the IUT that both fired the event and observed the event. These events are now persisted in xml files on their firing and updated during runtime validation. A complete set of reports are now created from the event files generated during the runtime validation process. In addition the use of extended ACL events allows the contract monitor to monitor the temporal grammar of the scenario independent of any external tool.

Fifth, we introduced time sensitive scenarios. Time sensitive scenarios are independent of inter-contract events and as such required special consideration. To address this issue we introduced the ACL clock and demonstrated how the clock can be configured to reflect the runtime validation requirements.

Finally, we have demonstrated how the proposed solution eliminates the need to write complex code from scratch to develop the contract monitors classes as presented in the JavaMOP solution.

3.8.1 Additional Considerations

Up to this point, we have discussed the primary issues that need to be addressed for the ACL Validation framework and presented a Java based solution. However there is one additional consideration to be mentioned. Previous research and proposals into the ACL/VF were implementation specific. For example the original framework was C# specific and was bound to a C# application. The alternative solution focused on JavaMOP and the underlying IUT was Java based. The proposed solution presented in this thesis is IUT independent. In effect, it is not required that the IUT be developed in the same language as the ACL/VF. Using Java Connector Classes (JCC) [90, 91] and Java's Service Oriented Architecture (SOA) [92, 93] the proposed solution can be bound to any IUT independent of the underlying software environment. This lies however out of the scope of this work and is left for future research.

4. Sample Walkthrough

In addition to developing a framework for contract monitoring, another critical element that arose when developing our solution; the mapping of an ACL contract to an executable object. Currently there is no tool that allows developers to map an instance of an ACL contract to an IUT. In our case study we present the framework that allows developers to textually create a set of classes that correspond directly to an ACL contract.

In order for an ACL contract to become an executable contract it must first be converted by the validation framework to a set of executable classes, or in our proposal a set of Java classes that can be executed by the JVM. The IUT and their corresponding methods are bound to these classes. Although the solution presented in this chapter is hand coded it can become the foundation for future research into the development of an ACL compiler or parser that will automatically convert an ACL contract to an executable class.

During our walkthrough we will elaborate how the various modules presented in Chapter Three collaborate with the ACL contracts to perform runtime validation. In addition, we propose advancements on the original contract and introduce new ACL type – the Dictionary. We will make notes of our modifications during our examination of the contracts and explain the reasoning behind the enhancements. For those specifications and requirements that are too long to include in the body of this thesis the code may be found at

https://drive.google.com/folderview?id=0B5cT_BjwahZrQWpiTl9iVXpFYW8&usp=sharing

You will also find there the source code and a complete set of log files generated by the proposed validation framework and other relevant artifacts.

4.1 University Case Study

In this section we focus specifically on the university management system developed by Arnold and Corriveau [94]. This case study is intensive in terms of responsibilities and scenarios and illustrates their more intricate semantics. In this section we focus primarily on the University contract (the main contract) and where required expand on the relationship with two other inter-related contracts, the Student and Course contracts. All three contracts are inter-dependent and required to drive a complete and exhaustive validation of the IUT

It is important to note that the ACL contracts were developed independent of this thesis. The contracts were implemented by the ACL/VF tool and were tested to ensure proper functionality. The IUT used to test our proposal was developed specifically for this thesis. However, the contracts used to model the IUT were the foundations for previous research in this area, and as such one can be assured that the issues addressed and solutions presented in this thesis are in sync with previous work. The IUT presented is exhaustive in functionality and presents a solid foundation from which to test our proposals. It must be noted that a complete and exhaustive study of the contracts presented is beyond the limits of this thesis and as such we will focus primarily on the aspects of the ACL/VF that is relevant to our work (namely scenario monitoring). We will note where functionality is limited or omitted and provide a brief explanation.

4.2 Objective

In this section we examine in detail the various semantics of an ACL contract including scenarios and present the related elements and functionality proposed for the VF.

The key ACL elements and concepts that need to be addressed in our mapping to the VF and monitoring components are the ACL scenarios, responsibilities and observabilities. When discussing scenarios there are some key features that need to be addressed; the lifespan of the

scenario, the temporal grammar of the scenario and the various paths a scenario a may take during runtime. This includes monitoring loop structures as well as the constraints imposed on the responsibilities defined in the scenarios grammar.

In addition, there are two additional ACL elements that need to be addressed, namely atomic sections and parallel blocks. An atomic section is restricted to those responsibilities defined in the atomic section, no other responsibilities from the contract may occur while execution remains in the atomic section. In effect, the atomic section must complete successfully before control is passed back the enclosing scenario. In parallel blocks multiple instances of the parallel sections may be active simultaneously, and each instance may contain its own path.

First we will examine the main contract – the University contract in detail and then examine the relevant sections of the Course and Student contracts.

4.3 University Contract

The following is the complete version of the university contract. In order to define the mappings to the Java contract classes, we divide the ACL grammar into blocks and present the Java grammar in corresponding blocks throughout our discussion.

4.3.1 Contract Definition and Instance Binding

The contract header identifies the contract as the main contract. The reserved word *main* indicates to the VF that the Station Manager is responsible for initializing instances of this contract and assigning them to the appropriate monitor station.

```
MainContract University <University>
```

The next line binds an instance of the IUT to the contract, specifically and instance of the university object to the contract. We must note that an instance of the contract is the *context* with

which the *instance* of the IUT is bound to. The instance of the IUT is bound to the contract by the Station Manager during the initialization process.

```
once Scalar tUniversity instance;
```

4.3.2 Variables and the Parameters Block

The next section to address is the Parameters block. The parameters block defines the contract variables and assigns their initial value. They are used to configure the contract and may be use specify different scenario paths. These values are determined at binding time and is handled manually. You will notice there are two types, the first is identified by the keyword *InstanceBind*. These parameters have contact scope and are bound to the current instance of the contract they are defined in. The keyword *InstanceBind* indicates the contract variable is bound to the IUT and must maintain the value set in the instance of the IUT, in effect each contact instance has its own value for that attribute. *InstanceBind* variables are final and cannot be changed during runtime. In addition contract-variable may be assigned a range and the value of the variable, when set during initialization must fall within these values.

```
Parameters
{
   [1 to 12]  Scalar Integer InstanceBind NumTermsToComplete;
   [1 to 100] Scalar Integer InstanceBind UniversityCourses;
   [1 to 100] Scalar Integer InstanceBind UniversityStudents;
   [1 to 100] Scalar Integer InstanceBind UniversityProfessors;

   Scalar Integer MaxCoursesForFTStudents = 4;
   Scalar Integer MaxCoursesForPTStudents = 2;
   Scalar Integer PassRate = 70;
}
```

The second set have global scope and are available to all contracts bound to the contract monitor. If a variable is not bound to the IUT it may be assigned a default value during initialization, however the IUT can override this value is not implicitly bound to the IUT. In the university contract, parameters are used to specify required properties such as the number of

courses, students and professors. Global scope variables are dynamic in nature and can be used to keep track of data independent of the IUT, that is, data relevant to the TRM. As such they can be used as a dynamic test oracle over the lifetime of the contract.

The following code corresponds directly to the ACL contract section we just illustrated. The sample code includes the section of the Java class that corresponds to the contract parameters. As we can see the source code matches closely to the ACL contract. As part of the initialization of the VF the Station Manager passes through an instance of the respective IUT object, in this case an instance of the University object and binds it to the contract. During the initialization of a contract instance the super class is initialized and assigns the contract to its respective monitor station and contract monitor and sets up the background threads required to manage the ACL observabilities (more on this later).

```
public class UniversityContract extends ACLContract
{

   private University instance;
   protected Parameters Parameters;

   @Parameters
   public class Parameters
   {
      private final int PassRate = ConfigManager.PassRate;
      private final int NumTermsToComplete =
                           ConfigManager.NumTermsToComplete;
      private final int MaxCoursesForFTStudents =
                            ConfigManager.MaxCoursesForFTStudents;
      private final int MaxCoursesForPTStudents =
                           ConfigManager.MaxCoursesForPTStudents;
      protected int UniversityCourses =
          UniversityHelper.getInstance().getRandomNumberOfCourses();
      protected int UniversityStudents =
          UniversityHelper.getInstance().getRandomNumberOfStudents();
      protected int UniversityProfessors =
          UniversityHelper.getInstance().getRandomNumberOfProfessors();
   }

   public UniversityContract (Object o)
   {
       super(o);
       instance = (University)o;
      Parameters = new Parameters();
   }
```

We must note at this point that in our proposal the ACL/VF contains an abstract class – *ACLContract* that all contract classes must inherit from. The *ACLContract* class contains the methods and attributes required to map a contract to the contract monitor and monitor station it is assigned to, along with the methods required to observe and fire ACLEvents.

During the initialization process the contract variables and parameters are configured by the IUT. As you notice the set of parameters are contained in an inner class and the parameters have a protected scope. *InstanceBind* variables, or contract variables are declared private and final. This insures that no other instance of the contract can access or modify the variables. The global variable are declared protected. This allows the parameters to be accessed locally and are visible to only those contract instances that are in the same scope (or contract monitor). This insures there can be no exchange of data between contract monitors, in effect making the variables context safe.

4.3.3 Observabilities

The Observability keyword denotes an observability block within the contract. Observability elements are read-only methods that are used to acquire state information from the IUT for evaluation within the contract. Contracts may contain any number of observability methods, however they and are limited in scope to the contract instance. Observability methods have the same lifespan as the contract and can be invoked at any point during the lifespan of the bound IUT instance.

```
Observability tDateTime CurrentDate();
Observability tTerm Term();

Observability List<tTerm> Terms();
Observability List<tCourse> Courses();
Observability List<tStudent> Students();
Observability List<tProfessor> Professors();

Observability Boolean RegistrationOpened();
Observability Boolean RegistrationClosed();
```

Example of Bound Observabilities from the University Contract

An observability can be implemented in one of two ways, either binding them directly to a method or set of methods in the IUT (a bound observability*),* or provide a body that performs a set of calculation and returns a value (unbound). The latter form may contain bound observabilities in its grammar.

```
@Observability(bind)
public Date CurrDate() {
   return getTimeStamp()[17];
}

@Observability(bind)
public Term Term() {
   return instance.getCurrentTerm();
}

@Observability(bind)
public List<Term> Terms() {
   return instance.getTerms();
}

@Observability(bind)
public List<Course> Courses() {
   return instance.getCourses();
}

@Observability(bind)
public List<Student> Students() {
   return instance.getStudents();
}

@Observability(bind)
public List<Professor> Professors() {
   return instance.getProfessors();
}

@Observability(bind)
public Boolean RegistrationOpened () {
   return instance.isRegistrationOpened(timeStamp);
}

@Observability(bind)
public Boolean RegistrationClosed () {
   return instance.isRegistrationClosed(timeStamp);
}
```

Example of the Java Code to Handle Bound Observabilities in the University Contract Class

[17] The *getTimeStamp* is a default method identified in the ACL/VF and is encapsulated in the ACL contracts abstract class.

4.3.3.1 Bound Observability

In our solution, a bound observability maps directly to a method that has the same name, signature and return type as defined by the observability in the ACL contract. The observability is bound to an IUT method that executes the required functionality. Currently this mapping is performed manually. You will notice the use of annotations. The annotation identifies the method in the contract class is an observability and the keyword *bind* denotes the observability type. We will discuss annotations in detail later in this chapter. Each observability is bound to the IUT through the *instance* variable bound to the IUT during the initialization of the contract instance. As can be seen, the set of observabilities match closely to their corresponding elements in the ACL contract. Each observability is bound directly to its related IUT method by the *instance* variable and returns the required state information from the IUT.

4.3.3.2 Unbound Observability

The next type of observability we address is the unbound observability. The following example of an unbound observability is taken directly from the course contract. The grammar defined in the body of an unbound observability is executed as a single statement and must define a return value; this is the default type returned by the observability (pg. 104-109). In addition, an observability may contain other observabilities defined in either the enclosing contract or an inter-related contract (as indicated in the following observability). The *each* statement is a loop statement that iterates[18] through the list of prerequisites returned by the *PreRequisites* observability (previously defined in the contract). In addition, an observability may contain pre and post conditions and check statements. However, unbound observabilities cannot contain invariants.

[18] An ACL *iterator* is a defined ACL type that is automatically type cast to the object type defined in the *list* statement in the variable declaration of the contract.

Also, an unbound observability may contain other ACL language elements including loop, each and choice statements. You will also notice that the ACL specificaiton defines a set of methods[19], for example *Length* and *Contains*.

```
Observability List <Integer> PreRequisites ();

Observability Boolean HasPreRequisites (tStudent student)
{
   value = true;
   Student s = bindpoint.student;

   List Integer completedCourses = s.CompletedCourses();

   Check(completedCourses.Length() not= 0);

   each ( PreRequisites() )
   {
      value = completedCourses.Contains ( iterator );
   }
}
```

In the preceding example, the observability must pass through the *check*[20] statement before it executes the *each* statement. If it fails the *check* statement an appropriate runtime error message is sent to the CER and the observality stops exectution and returns false.

```
@Observability(bind)
public List<Course> PreRequisites ()
{
   return instance.getPreRequisites();
}
```

In the following sample code from the course contract class you will notice we pass and instance of the student contract as a parameter. Contract classes are designed to interact with other

---

[19] A complete list of these predefined methods is outside of the scope of thesis, as such we will only address those methods that are directly related to our thesis.

[20] Pre and post conditions, check statements and invariants are now built into the ACL/VF and will be discussed in detail later in this chapter

contract classes and not to interact directly with IUT instances not bound to the containing contract class. In other words, they can only access the instance of the IUT that is bound directly to the current contract class instance. Pre and post conditions and check statements are converted to Java Assert[21] statements.

```
@Observability(unbound)
public Boolean HasPrequisites(StudentContract tStudent)
{
   Boolean value = true;

   Student s = tStudent.getInstance();
   List<Course> completedCourses = s.CompletedCourses();

   Assert(completedCourses.size() != 0);

   for (Course o: PreRequisites())
   {
      value = completedCourses.contains(o);
   }
   return value;
}
```

Recall that a contract instance has an IUT instance bound to it. The ACL/VF abstract contract class has a defined method that returns the instance of the IUT bound to the contract. In the above example, the *getInstance* method invoked on the student contract returns the instance of the IUT student class bound to the student contract. This requirement maps directly to the ACL keyword *bindpoint*. You will also notice the each statement maps directly to the Java for statement.

4.3.3.3 The Observe Statement

The block below shows an alternate observability element that is not covered in the ACL specifications. This type of observability is dependent upon the observation of an event before it queries the IUT. The observability is dependent upon the idea of the ACL *observe* element. The issue that arose that drove this new implementation was that certain elements of a scenario may be

[21] Java Assert statements are similar in nature to Junit assertions. With Java asserts we can define specific error messages, which would map directly to ACL belief statements. We will discuss this feature in detail later in this chapter

time sensitive and as such the firing of the observable events are time dependent. Since the temporal grammar must be honored if the scenario is to complete successfully the temporal grammar is dependent upon the observation these events. This requires a background thread (or listener) be invoked that will wait until the event is triggered, and when observed return control to the scenario. You will notice that the observability defined in the ACL contract defines the event to observe and contains to parameters or defines an observability method (no closing parenthesis)

```
Observability Boolean TermStarted;
Observability Boolean TermEnded;
Observability Boolean LastDayToRegister;
Observability Boolean LastDayToDrop;
```

Example of Event Specific Observabilities

The following is an example of the Java code generated to map the observability to the contract class that maps to the above observabilities. The Observe statement is encapsulated in the abstract contract class and is now contains in the ACL/VF, more on this later.

```
@Observability(observer)
public Boolean TermStarted()
{
   Observe (TermStarted,instance);
   return instance.getTermStarted(timestamp);
}

@Observability(observer)
public Boolean TermEnded()
{
   Observe (TermEnded,instance);
   return instance.getTermEnded(timestamp);
}

@Observability(observer)
public Boolean LastDayToRegister()
{
   Observe (LastDayToRegister,instance);
   return true;
}

@Observability (observer)
public Boolean LastDayToDrop()
{
   Observe (LastDayToDrop,instance);
   return true;
}
```

For example, the *Term* scenario in the university contract contains the observability *TermStarted* in its grammar to determine if the current term the scenario is running against has started. This event is time sensitive in that the event must be fired by either 1) an inter-related contract or 2) an external process (i.e., the clock).

When the observability is invoked it spawns a background thread that queries the Blackboard until the event is observed. If the scenario terminates before the event is observed, the temporal grammar of the scenario is considered invalid and logged to the CER. These observabilities are defined in the abstract contract class and are part of the VF. In the following code segment you will notice it is a simple do-while loop that terminates when the ACL event has been observed. If the contract terminates before the event is observed the thread is still active in the thread group and an error report is generated upon termination of the contract instance in the finalize statement.

```
protected Boolean Observe (ACLEvent event, ACLContract context)
{

   Thread t = new Thread(tg);

   public void run ()
   {
      do {
        try {
            t.wait();
          }catch(InterruptedException e){
            e.printStackTrace();
          }
      } while (!Blackboard.getInstance().observe (event,context));
   }
   t.start();

   return true;
}
```

The parameter list must contain the ACL event to observe and the current instance of the contract (the context). We have extended this observability to include two or more events and create a compound AND or OR statement. In effect we can monitor for the firing two or more events simultaneously.

4.3.4 Responsibilities

The next ACL element to discuss is the responsibility. The responsibility keyword denotes a responsibility block in a contract. A contract may contain any number of responsibilities. A responsibility can be viewed as a functional requirement or a task. Responsibilities can be defined in one of two ways. First, it can be bound directly to an IUT method or group of methods (a bound responsibility) in the IUT, or we can define the responsibility through the use of execution grammar that in turn uses other responsibilities (unbound).

Regardless of the type, responsibilities are denoted by a method like signature: optional modifiers, a return type, name, and optional parameters. If a responsibility does not define a return type, the return type Void[22] by default. A responsibility may contain any number of pre and post conditions and ACL checks, including invariants. In addition, a responsibility must contains the *Execute*[23] statement. The execute statement denotes where in the grammar the actual execution of the corresponding IUT method or methods are to occur. The pre and post conditions are associated directly with the execute statement. All pre-condition must occur before the execute statement and all post-conditions must occur after the execute statement. These pre-and post-conditions check the state of the IUT before and after execution.

We must note there are two specialized responsibilities; *new* and *finalize*. Since new and finalize are reserved words in Java we modified the syntax of both responsibilities in the Java source code. The *new* responsibility is responsible for evaluating the IUT before the creation of the contract instance, the *finalize* responsibility is responsible for determining the state of the IUT after the contract instance is destroyed. Both these responsibilities are managed by the contract instance during the creation and destruction of their associated IUT instances.

[22] Void is an ACL defined scalar type
[23] This is also the binding point in the contract class

```
Responsibility new()
{
    Post(Courses().Length() == 0);
    Post(Students().Length() == 0);
    Post(Professors().Length() == 0);
}
```

Example of a new Responsibility

```
Responsibility new()
{
    Post(Courses().Length() == 0);
    Post(Students().Length() == 0);
    Post(Professors().Length() == 0);
}
```

Example of a finalize Responsibility

The following is an example of the corresponding Java source code to handle the defined responsibilities identified above. You will notice we have defined a Java Responsibility annotation and a set of related annotation types.

```
@Responsibility(new)
public void New()
{
    //Post Conditions
    Post(Courses().size() == 0);
    Post(Students().size() == 0);
    Post(Professor().size() == 0);
}
```

Example of a Java new Responsibility Source Code

```
@Responsibility(finalize)
public void Finalize()
{
    //Pre Conditionss
    Pre (Courses().size() != 0)
    Pre (Students().size() != 0);
    Pre(Professor().size() != 0);
}
```

Example of a Java finalize Responsibility Source Code

4.3.4.1 Bound Responsibilities

Our next illustration is a bound responsibility. The example is taken directly from the university contract implemented in the university case study. The responsibility takes a Course as a parameter and has no return type. The responsibility has a pre-condition to test whether the course

is currently registered with the university, and a post condition the test whether the course is successfully removed from the course list. The *Execute* statement is used to control when a method bound method from the IUT is invoked. The following are the set of course related responsibilities defined in the university contract. The set of student and professor responsibilities are identical in nature and require no further elaboration.

```
Responsibility CreateCourse(String name, Integer cap)
{
   once tCourse value;
   once Scalar Integer oldSize;
   oldSize = PreSet(Courses().Length());

   Execute();

   Post(value.Name() == name);
   Post(value.CapSize() == cap);
   Post(Courses().Length() == oldSize + 1);
   Post(Courses().Contains(value) == true);
}

Responsibility CancelCourse(tCourse course)
{
   Pre(Courses().Contains(course) == true);
   Execute();
   Post(course.Cancelled() == false);
}

Responsibility DestroyCourse(tCourse course)
{
   once Scalar Integer oldSize;
   oldSize = PreSet(Courses().Length());

   Pre(Courses().Length() > 0);
   Pre(Courses().Contains(course) == true);

   Execute();

   Post(Courses().Length() == oldSize - 1);
   Post(Courses().Contains(course) not= true);
}
```

Example of Bound Responsibilities

The *CreateCourse* and *DestroyCourse* are the more complex responsibilities and highlights some additional features of ACL. The responsibilities take a course as a parameter and declares no return type. In addition they contain post conditions that ensure the course has either been added

or deleted successfully by checking the current size of the course list with the previous size of the course list.

It is important to note that ACL contains a defined set of attributes that define ACL data types. For example the term *Scalar* denotes an ACL scalar type, in this case an Integer, *once* indicates that variable can only be assigned a value once and is final. In addition ACL defines a set of methods may be invoked on scalar type, in this example *Length* returns the number of elements in the Array and *Contains* returns an ACL Boolean type indicating if the Array contains the actual element under consideration. The *PreSet* notation in ACL defines an assignment of a value during pre-condition evaluation and is used in the subsequent post-condition. The following is an example of mapping a bound responsibility to the Java source code.

```
@Responsibility(bound)
public void CreateCourse (String name, Integer cap)
{
   int oldSize = Courses().size();
   Course value = (Course) Execute (instance.createCourse (name, cap) );

   Post (value.Name() == name );
   Post (value.CapSize() == cap);
   Post (Courses().size() == oldSize + 1);
   Post (Courses().contains(value));
}

@Responsibility(bound)
public void CancelCourse(Course tCourse)
{
   Pre (Courses().contains(tCourse) == true);
   Execute (instance.cancelCourse(tCourse) );
   Post (tCourse.Cancelled() == true);
}

@Responsibility(bound)
public void DestroyCourse(Course course)
{
   int oldSize = Courses().size();
   Pre (Courses().size() > 0);
   Pre (Courses().contains(course) == true);
   Execute (instance.destroyCourse (course) );
   Post (Courses().size() == oldSize - 1);
   Post (Courses().contains(course) == false);
}
```

Example of Mapping an ACL Responsibility to Java Source Code

A bound responsibility does not have to return a type and may be a simple responsibility that is bound the IUT to perform a specific function. They are primarily used by other responsibilities. The next three responsibilities provide examples. You will notice the simple responsibility type contains no body or grammar. The choice statement, each statement and iterator will discussed in the next section.

```
Responsibility CalculatePassFail()
{
   each(Students())

   choice(iterator.failures) >= 2)
      FailStudent(iterator);
   alternative
      PassStudent(iterator);
}

Responsibility FailStudent(tStudent student);
Responsibility PassStudent(tStudent student);
```

Example of Bounded Responsibilities

The following is an example of mapping a simple responsibility to the corresponding Java code.

```
@Responsibility(unbounded)
public void CalculatePassFail()
{
   for (Student iterator: Students())
   {
      if (iterator.getFailures() >= 2)
         FailStudent(iterator);
      else
         PassStudent(iterator);
   }
}

@Responsibility(bind)
public void FailStudent(Student student)
{
   instance.setFailStudent(student);
}

@Responsibility(bind)
public void PassStudent(Student student)
{
   instance.setPassStudent(student);
}
```

Example of Mapping Simple Bounded Responsibilities to Java Source Code

The next responsibility is an example of how a parameter or global variable is used.

```
Responsibility ReportMark(tStudent student, Integer mark)
{
   choice(mark) < Parameters.PassRate)
   {
      student.failures = student.failures + 1;
   }
   Execute();
}
```

The parameter *PassRate* is defined in the Parameter block, as previously discussed, and has global scope, that is it is available to all contracts bound the same contract monitor, in this case the university contract monitor.

```
@Responsibility(bind)
public void ReportMark(Student student, Integer mark)
{
   if (mark < Parameters.PassRate)
   {
      Execute (student.setFailure());
   }
}
```

4.3.4.1 Unbound Responsibilities

In the previous example we provided an example of an unbound responsibility; *CalculatePassFail*. The example illustrates how an ACL *iterator*, in this case the *each* statement loops through a list, determined by the *Students* observability. The *choice* statement determines if the student passes or fails.

```
@Responsibility
 public void CalculatePassFail()
 {

   for (Student iterator: Students())
   {
     if (iterator.getFailures() >= 2)
        FailStudent(iterator);
     else
        PassStudent(iterator);
   }
}
```

The keyword *iterator* refers to an instance of the Student contained in the list returned by the *Students* observability. The unbound responsibility is reliant upon two other responsibilities define in the contract to successfully complete the grammar.

Note the close mapping between the ACL *each* statement and the Java *for* statement. Also note the mapping between the ACL *choice/alternative* statement and the corresponding Java *if/else* statement. In addition, the example illustrates how close we adhere to Java's naming conventions and the use of *getter* and *setter* methods. Depending on the result of the *choice* statement the corresponding bound responsibility is invoked. Note the *iterator* term and type of the iterator, the mapping ACL grammar and Java code is one to one.

4.3.5 Scenarios

So far in this section we have shown how contract classes closely match the ACL semantics and syntax for parameters, contract variables, responsibilities and observabilities. The next examples are the most significant of this thesis, ACL scenarios. These examples show the complexities of ACL scenarios and how they may be handled with our proposal. The two main scenario discussed in this section are the *Term* scenario and the *TerminateProfessor* scenarios.

Note the use of the + modifier in the *Term* scenario. The modifier indicates that the scenario must run one to many times. This is indicative of a *TimeThread*; an instance of a thread based scenario. For the purpose of this thesis we define a main scenario as the principal scenario that governs the sequence of events based on temporal grammar. Through its grammar it determines when the responsibilities and observabilities are invoked, and through the firing and observation of events controls the initialization and termination of other sub-scenarios, either locally or externally.

In this scenario we introduce a new ACL element – the *TimeThread.* A TimeThread indicates that a scenario is time sensitive and as such the firing and monitoring of events have temporal grammar associated with them. The term *TimeThread* is borrowed from Buhr and Casselman's work on Use Case Maps in their book Use case Maps for Object-Oriented Systems [45].

```
Scenario Term
{
   Trigger(new()),
   TimeThread,
   (
    CreateCourses(dontcare, dontcare)[Parameters.UniversityCourses],
    CreateStudents(dontcare)          [Parameters.UniversityStudents],
    CreateProfessors(dontcare)        [Parameters.MaxProfessors],

    AssignProfessorsToCourses(dontcare,dontcare)
                                  [Parameters.UniversityCourses],
    TermStarted(),
    fire(TermStarted,Term()),
    LastDayToRegister(),
    fire(LastDayToRegister,Term()),
    LastDayToDrop(),
    fire(LastDayToDrop,Term()),
    TermEnded(),
    fire(TermEnded),
    observe(MarksRecorded)            [Parameters.UniversityCourses],
    CalculatePassFail(),
    DestroyCourses(dontcare)          [Parameters.UniversityCourses],
    DestroyStudents(dontcare)         [Parameters.UniversityStudents],
    DestroyProfessors(dontcare)       [Parameters.UniversityProfessors],
    fire(TermComplete,Term());
   )+,
   Terminate(finalize());
}
```

Term Scenario

The scenarios grammar defines the sequence of responsibilities and observabilities required for a universities term to be completed successfully. It handles the creation of courses, students, professors and assigns professors to the courses. It determines when the term has started and when the term has ended, when registration is opened and students can register for courses, the last day for registration, the last day to drop courses, and when the marks are recorded. In addition when a term ends it cleans up the courses, students and list of professors. Upon completion of a term the scenario fires an event to notify the other related contracts the term has ended. If the exit condition

is met the scenario terminates and shuts down. If the exit condition, or termination event has not been observed the scenario proceeds to the next term and repeats the sequence of events.

As we will illustrate, the observabilities contained within the grammar of the scenario determines the lifespan of both the course and students contract instances and the events fired will determine when the scenarios in the inter-related contracts are both initialized and terminated.

```
@Scenario(timethread)
 public class Term extends TimeThread
 {
    protected int term = 1;

    public Term (ContractMonitor monitor) {
       super(monitor);
    }

    @Override
    public void run()
    {
       Trigger(New);
       do {
          CreateCourses(dontcare,dontcare);
          CreateStudents(dontcare);
          CreateProfessors(dontcare,dontcare);
          AssignProfessorsToCourses(dontcare,dontcare);
          TermStarted();
          Fire(TermStarted,Term());
          LastDayToRegister();
          Fire(LastDayToRegister,Term());
          LastDayToDrop();
          Fire(LastDayToDrop,Term());
          TermEnded();
          Fire(TermEnded, Term());
          Observe(MarksRecorded);
          CalculatePassFail();
          DestroyCourses(dontcare);
          DestroyProfessors(dontcare);
          DestroyStudents(dontcare);
          Fire(TermCompleted,Term());
       } //+
       while(!done);
       Terminate(Finalize);
    }
 }
```

Mapping a Scenario to Java Source Code

The first thing you will also notice it is an inner class. This ensures the scenario is local to the contract instance it is bound to and cannot be initialized unless an instance of the enclosing contract

has been instantiated. In addition the scenario extends the *TimeThread* class and contains the Java threads *run* method. The *TimeThread* and *ACLThread* classes are abstract classes that are built into the ACL/VF and encapsulates the methods required to initialize and manage scenarios and scenario threads.

```
public abstract class TimeThread extends ACLThread
{
   protected ACLTimeThread (ContractMonitor monitor)
   {
      super();
      // Set to the Current ACL Contract
      super.setContext(monitor.getContext());
      // Set to the Current Contract Monitor
      super.setContractMonitor(monitor);
      // Set to the Current Monitor Station
      super.setMonitorStation(monitor.getMonitorStation());

      super.event = EventFactory.getInstance().getEvent(NullEvent,this);

      BlackBoard.getInstance().addObserver(this);

      ACLEvent e = EventFactory.getInstance().getEvent(ThreadStarted,this);
      e.save();
    }
}
```

For example, the *CheckBreakPoint* method which called by the *Terminate* condition defined in the ACL scenario is defined in the *TimeThread* class.

```
protected void Terminate (ACLEvent event)
{
   if (CheckBreakPoint())
   {
      BlackBoard.getInstance().Fire (Terminate, this);
   }
}
```

Example of Mapping the Terminate Statement to Java Source Code

```
protected void checkBreakPoint ()
{
   if (condition == ConfigManager.getInstance().breakPoint())
   {
      done = true;
      return;
   }
   condition += 1;
}
```

Java Source Code Breakpoint Condition

If the terminate condition for the scenario is met the *CheckBreakPoint* sets the terminate flag to true, fires a *Terminate* event to the Blackboard and the scenarios thread ends, else the thread returns to the starting point. Upon termination the *Terminate* method in the parent class is called and the *finalize* responsibility is invoked and the thread is cleaned up. There is a process in the contract monitor that listens for a *Terminate* event. When a *Terminate* event is observed it invokes the finalize method in the corresponding contract instance.

4.3.5.1 Scenarios and TimeThreads

The following section will walk through a time sensitive scenario and describe how *TimeThread* is implemented. First we have a triggering condition, the condition or the state the IUT must be in before the scenario is instantiated. In this example the *TimeThread* waits for a *new* event to be fired indicating a new university has been created and bound to the monitor station. Once this condition is met the *new* responsibility is called, the state of the IUT is verified and the thread is started. The second aspect of the structure is that the thread contains a simple do-while loop. It relies on an event to occur (or fired) before entering the do loop and an event to be fired to terminate the loop.

The *TimeThread* class encapsulates the methods required to monitor triggering events and termination conditions. In addition when a scenario is initialized by the containing contract monitor, the station manager, the context (contract) the scenario is bound to and the instance of the IUT is bound to the scenario. In addition each scenario has its own thread id and assigned to contracts thread group.

The Java Virtual Machine (JVM) allows an application to have multiple threads running concurrently. Every thread is assigned a priority by the JVM when initialized. Threads with higher priority are executed in preference to threads with lower priority. When code running in a thread

spawns a new thread, the new thread has its priority initially set equal to the priority of the spawning thread. However, Java allows the threads priority to be assigned during runtime. In effect, this allows us to dynamically change the priority of the threads in a thread group based on the current state of the IUT and the scenarios. For example, the *Term* scenarios thread and related observability threads are assigned to the same thread group. The *TimeThread* (scenario) has the highest priority and the current observability has the second highest priority, all dormant threads are assigned the lowest thread priority.

Since the main scenario has temporal grammar, the active thread is confined to the current observable event, past and future observable event threads are placed on a wait list, i.e. sleep, and invoked when the temporal grammar has identified their next in the scenarios grammar. The scenario's monitoring thread yields (or passes) control to a secondary thread that controls the execution of the observability currently identified in the scenarios grammar and waits (sleeps) for the observable event to complete and return a status of true (observed). Upon completion of the secondary thread, the main thread (the scenario thread) is activated and control is returned to the main thread. The secondary thread is placed back on the waiting list and sleeps. At any one time during the lifespan of the scenario only one thread is active, all other threads in the group are put on a wait list and become dormant. Assigning a thread name and any related data, or thread specific information is optional.

4.3.5.2 Temporal Grammar

Now we will discuss the temporal grammar and responsibilities of the scenario in detail. The first event to occur is the *new* event. It triggers the entrance into the do loop. The scenario grammar contains a set of secondary scenarios in a sequential order. For the purpose of our thesis we define a secondary scenario as a scenario that is called from the primary or main scenario. These

secondary scenarios map directly to methods inside the contracts related Java classes. For example, the following ACL code defines a set of secondary scenarios that are called from within the main scenario. Notice the use of + operator. This indicates the responsibility may be called multiple times.

```
Scenario CreateCourses
{
   Trigger(new()),
   CreateCourse(dontcare, dontcare)[Parameters.UniversityCourses],
   Terminate(fire(CoursesCreated));
}

Scenario CreateStudents
{
   Trigger(new()),
   CreateStudent(dontcare)+,
   Terminate(fire(StudentsCreated));
}
```

The following code snippets illustrate a secondary or sub-scenario in the contracts Java related code. The CreateProfessors scenario is identical in nature and will not be discussed any further.

```
@Scenario
private void CreateCourses (Object ... o)
{
   Trigger(New);
   do
   {
      CreateCourse(null,null); // Calls a Responsibility
   }
   while (Courses().size() < Parameters.UniversityCourses);
   Terminate(Fire(CoursesCreated));
}

@Scenario
private void CreateStudents (Object ... o)
{
   Trigger(New);
   do
   {
      CreateStudent(dontcare); // Calls a Responsibility
   }
   while (Students().size() < Parameters.UniversityStudents);
   Terminate(Fire(StudentsCreated));
}
```

A secondary scenario has a trigger condition and a termination condition. Once terminated the ACL/VF fires an event to the blackboard indicating the scenario is completed and control is

returned to the main scenario. Inside the scenario do-while loop a contract responsibility or set of responsibilities are invoked and passed the required parameters. The keyword *dontcare* indicates that during runtime the contract has no restrictions on the parameter values passed in. In essence we are left to randomly create the desired objects or data elements. The *dontcare* keyword maps directly to the Java *null* object.

The next set of statements implement a sub-set of the contracts observability methods, the time sensitive observabilities. As we proceed through the temporal grammar of the scenario the first time sensitive event is *TermStarted*. The TermStarted observability runs a background thread and waits for the *TermStarted* event to be fired. This event is triggered by the ACL clock, the IUT instance uses the timestamp fired by the clock to determine if a term has started. Recall the ACL clock has a start date and end date. For the purposes of this thesis the clock was set to advance twenty four hours every 1000 millisecond. As the clock advances and changes dates the Blackboard is notified of the change. In addition each contract monitor is registered with the clock and as the clock advances they are notified of the date change. The date changes are pushed to the contract and the contract observabilities use these timestamps to query the IUT. However, the timestamp is an attribute of the abstract ACLContract class and is not an attribute of the contract instance itself (unless otherwise defined directly in the contract). Once an observability is invoked and the event is observed by the observability (sub-scenario) the observability terminates and control is returned to the main thread (main scenario) and continues to the next event in the grammar and the process is repeated until completion of the main scenario.

The next element in the temporal grammar is the *Observe (MarksRecorded)* statement. At this instant in the temporal grammar the scenario waits for the course contract to fire the *MarksRecorded* event. Upon observation of the *MarksRecorded* event the scenario executes the

CalcuatePassFail responsibility. Upon completion the scenario continues and cleans up the data associated with the current term (i.e. *DestroyCourses*, *DestroyStudents* and *DestroyProfessors*).

```
@Scenario
private void DestroyCourses (Object ... o)
{
    Integer size = Courses().size();

    Trigger(TermCompleted);
    do
    {
        DestroyCourse(null);
    }
    while (Iterator() < size);
    Terminate(Fire(CoursesDestroyed));
}

@Scenario
private void DestroyStudents (Object ... o)
{
    Integer size = Students().size();

    Trigger(TermCompleted);
    do
    {
        DestroyStudent(null);
    }
    while (Iterator() < size);
    Terminate(Fire(StudentsDestroyed));
}
```

Once the scenario has cleaned up the data associated with the current term the *TermCompleted* event is fired. The events observed and fired throughout this scenario are parameterized events. Along with the ACL event type, they pass through the current term the event is associated with. This has two benefits, first it allows detailed information to be sent the CER and second it allows for refinement of event monitoring by other contracts. This ensures that scenarios in corresponding contract instances are in sync with the principal scenario.

4.3.5.3 Terminate Professor Scenario

The last scenario to discuss with respect to the University contract is the *TerminateProfessor* scenario. This scenario illustrates the use of a parameterized thread and an alternate method to define the termination condition.

```
Scenario TerminateProfessor
{
   Trigger(observe(u.TermStarted)),
   TimeThread,
   {
      observe(TerminateProfesso(tProfessor)),
      DestroyProfessor(tProfessor),

      each (Courses()
      {
          choice(iterator.professor) == tProfessor)
         {
            CancelCourse(iterator),
            fire(CourseCancelled(iterator));
         }
     }
     fire (ProfessorTerminated(professor));
   } [observe(u.TermEnded)];
}
```

Terminate Professor Scenario

The event signaling the scenario to terminate a professor is fired by a background process and is independent of any ACL contract. The background process is initialized when the ACL/VF is invoked and registered with the clock. The VF is responsible for the initialization of both the clock and the background processor.

```
private class TerminateProfessorProcess implements Events, Runnable
{
@Override
public void run() {
   do
   {
      if (terminateProfessor())
      {
         Professor p = UniversityHelper.getInstance().getRandomProfessor();
         Fire(TerminateProfessor,p);
      }

      try
      {
        tpThread.wait(waitTime);
      } catch (InterruptedException e)
      {
        LOGGER.info(e.getMessage(), e);
      }

   } while(!done);
   Terminate("TerminateProfessorProcess");
}
```

Terminate Professor Background Process

The background process randomly selects a day and a professor to terminate and fires the parameterized event *TerminateProfessor* event to the Blackboard with the professor as the attribute. Similar action and events are performed for the terminate student scenario.

```
private boolean terminateProfessor ()
{
   Random rand = new Random();
   int i = rand.nextInt(100);
   return i < terminationRate ? true : false;
}
```

Randomizing Terminate Professor

The structure and initialization of the scenario is reflective of the previous scenario discussed. The key features illustrated are 1) the scenario is triggered when a term is started and 2) terminates when a term is ended. The logic behind this is there is a period between terms when the termination of professor, or student would not affect the scenarios paths. The courses have to be created prior to the creation and assignment of professors. In effect the termination of a professor would have no impact before a term is started and after a term is completed. The Observe statement illustrates an ACL parameterized event. The thread waits until a *TerminateProfessor* event is fired and captures the professor bound to the ACL event. It then proceeds to destroy the professor by invoking the *DestroyProfessor* responsibility, passing the professor through as an attribute. Upon completion of the responsibility the scenario loops through the courses assigned to the university and cancels the courses assigned to the professor by invoking the *CancelCourse* responsibility. Note the value assigned to the iterator is passed through to the responsibility. The scenario fires the *CourseCancelled* event and passes the course as a parameter. This is observed by the Student contract and the related courses are dropped by the student contract. Once the courses are cancelled the *CoursesCancelled* event is fired signaling the termination of this section of the grammar, however the scenario is not terminated but waits until another *TerminateProfessor* event is fired or the termination of the scenario is signaled.

```
@TimeThread
public class TerminateProfessor extends TimeThread
{
   public TerminateProfessor (ContractMonitor monitor) {
      super(monitor);
   }

@Override
public void run() {
   Trigger(TermStarted);

   do {
      Observe(ProfessorTerminated(tProfessor));
      DestroyProfessor(tProfessor);

      for (Course c: Courses()) {
         if (c.getProfessor().equals(tProfessor)) {
            CancelCourse(c);
            Fire(CourseCancelled,c);
          }
      }
      Fire(CoursesCancelled,tProfessor);
   } while(Observe(TermEnded,false));
}
```

Mapping the Terminate Professor Scenario to Java Code

Note this scenario does not implement the repeat operators "+" (one or more) or "*" (zero or more) scenario modifiers. As such the termination condition reflects the observability defined in the ACL contract. Only the "+" or "*"modifiers are configurable and managed by the *CheckBreakPoint* method in the ACL/VF.

In this section we discussed mapping ACL contracts to their Java related classes and introduced specialized observability, more specifically a time sensitive observability that controls the flow of the scenarios grammar and related temporal grammar. In summary we have shown how our solution provides mapping from ACL contracts and scenarios to a set of Java classes. In addition we have illustrated how our proposal hides the complexity of scenario monitoring and extrapolates it from the contract classes. In addition the proposed mappings adheres as close as possible to the original ACL syntax and semantics and provides direct mapping between the two. We will now examine the Course contract.

4.4 Atomic Statements and Parallel Blocks

As previously mentioned, an atomic section in ACL indicates that only those responsibilities defined within the atomic segments grammar and the temporal grammar are considered valid. That is, the contents of the atomic element are to be viewed as an atomic unit from the scenarios point of view. When not in the atomic section, any responsibility not directly specified in the scenario may occur without triggering a violation of the scenario. However, in an atomic section if any responsibility from the contract, or set of contracts occurs, they are checked against the atomic statements grammar.

A further constraint on the atomic section comes directly from the ACL specification [95] "Atomic elements cannot be nested nor can they contain parallel elements". It is assumed that that the ACL specification document is correct and therefor the nested parallel statement in the original contract becomes a critical issue in our thesis.

The example above is complex in nature and as such we will focus primarily only on those elements central to our thesis. The first element of interest is the *Trigger* statement. You will notice that the *Trigger* takes a parameterized observability as its parameter. This requires that once the courses have been created and registered with the university contract, the university contract fires the *CourseCreated* event. The *IsCreated* parameter executes the corresponding observability defined in the student contract. The observability returns a Boolean value indicating if the current student contract this scenario is bound to is created and exists in the university contract. Remember that each instance of a student has a unique id assigned to it when initialized by the JVM – *the hashcode.*

```
Scenario RegisterForCourses
{
   Scalar tCourse course;
   Contract University u = instance;

   Check(u.RegistrationOpened()),
   Trigger(observe(CoursesCreated), IsCreated()),
   choice(IsFullTime())
   {
      (
        atomic
        {
           course = SelectCourse(u.Courses()),
           choice(not course.IsFull() | course.IsCancelled())
           {
              course = SelectCourse(u.Courses()),
              redo;
           };
         };
         u.RegisterStudentForCourse(context, course),
         RegisterCourse(course);
      )[0-u.Parameters.MaxCoursesForFTStudents] | Cancelled();
   }
   alternative
   {
      (
         atomic
         {
            course = SelectCourse(u.Courses()),
            choice(course.IsFull() | course.IsCancelled())
            {
               course = SelectCourse(u.Courses()),
               redo;
            };
         },
         u.RegisterStudentForCourse(context, course),
         RegisterCourse(course);
      )[0-u.Parameters.MaxCoursesForPTStudents] | Cancelled();
   },
   Terminate();
}
```

Example of an Atomic Statement from the Student Contract

Another important element to notice is that the *Check* statement requires that the university contract is bound to the current student contract. This is accomplished through the binding method described in the university contract walkthrough.

The *choice* statement comes next. In the original ACL syntax the choice statement takes a parameter and evaluation phrase. For example in the course contract the *AddStudent* responsibility has the following *choice* statement:

```
choice(Parameters.EnforcePreRequisites) true
{
   AddStudentPreReqCheck(s);
}
alternative (false)
{
     AddStudentNoPreReqCheck(s);
}
```

Syntax of the Original ACL Choice Statement

We propose the syntax of the choice statement be revised and drop the evaluation phrase; in effect it will behave like a switch statement. The first comparator in the choice statement, must evaluate to true to pass through else it executes the alternative statement. The modification to the ACL syntax would allow for complex choice statements to be developed and implement compound choice statements **(and, or)** statements as illustrated in the atomic statements in the scenario defined above. For example the previous choice statement would be rewritten as:

```
choice(Parameters.EnforcePreRequisites)
{
   AddStudentPreReqCheck(s);
}
alternative
{
   AddStudentNoPreReqCheck(s);
}
```

Proposed Syntax for the ACL Choice Statement

The next element is the atomic block itself. Remember an atomic block is self-contained and the elements listed in the grammar must operate in the sequence defined by the grammar. The atomic block illustrated is responsible for managing a student as they register for their courses. The section of the choice statement executed is dependent upon the student's status, i.e. full time or part time. For this we must note than in or test cases when we create an instance of a student

contract we randomly assign their status. This way the student's status may vary from one term to another and as such variations on the scenarios path may occur.

```
@Scenario(atomic)
public class RegisterForCourses extends Scenario {

public RegisterForCourses(Object o, String name) {
   super(o, name);
}

@Override
public void run() {
   CurrentCourses().clear();

   Course tCourse;
   UniversityContract u = (UniversityContract) getBaseMonitor().getContext();

   Check(u.RegistrationOpened());
   Student tStudent = (Student)instance;

   Trigger(Observe(CoursesCreated), IsCreated());
   if (IsFullTime()) {
      do {
         atomic (FullTime);
         {
            tCourse = SelectCourse (u.Courses());
            if (tCourse.isFull() || tCourse.isCancelled())
            {
               do {
                  tCourse = SelectCourse (u.Courses());
               }
               while (tCourse.isFull() || tCourse.isCancelled());
            }
         }
         u.RegisterStudentForCourse(tStudent, tCourse);
         RegisterCourse(tCourse);
      } while (
            tStudent.getCurrentCourses().size() <
                                       u.Parameters.MaxCoursesForFTStudents
            || Cancel()
      );
   }
   else { ... }
   Terminate (tStudent);
}
```

Mapping and ACL Atomic Statement to Java Code

During execution of the atomic block the student selects a course to register for, in our test cases this is a random selection from the current course list assigned to the university. The choice statement determines if the selected course is full or if it has been cancelled (identified

corresponding methods listed in the choice statement). Notice in the modification to the ACL syntax and the introduction of the *OR* operator. If either of these statements are true the process is repeated until a valid course is selected. Note that we use the ACL built in element - the *redo* statement. The *redo* element is analogous to the *continue* statement in Java and returns the program to the top of the *atomic* structure.

Upon successful selection of a course the scenario executes the *RegisterStudentForCourse* responsibility by universities contract bound to the student contract, passing an instance of the current student contract and the course selected as its arguments. The student contract then proceeds to invoke the *RegisterStudent* responsibility Execution of the atomic statement continues until either the maximum number of courses the student can register for has been reached or the student manually cancels the registration process.

The *Cancel* statement, or observability is also introduced here. The observability statement queries the IUT during runtime to determine is the student has requested to cancel the registration process. The functionality behind the *Cancel* observability is built into VF and triggers a break in the atomic statement and exits the atomic statement completely. The state of the IUT prior to the Cancel process Since the structure of the scenario is similar to the time threads illustrated in the university contract walkthrough we will limit our explanation to the corresponding Java code responsible for managing the atomic statements. In addition since the ACL grammar and source are similar for both the part time and full time students we limit the code snippet to full time students.

An atomic statement is similar to a time thread in that a background thread monitors and controls the execution of the grammar. The lifecycle of the atomic statement is managed by the contract monitor.

The only additional feature to mention is that in the atomic statement is that we pass through the condition that triggered the atomic statement to the contract monitor. The thread is responsible for ensuring the grammar of the atomic statement is met and the atomic statement is executed as a whole and logging all events. The following section of codes illustrates mapping an atomic statement to its corresponding Java code Notice we specify a new annotation type, atomic. In the following code segment we leave out the else statement since the *alternative* statement is identical to the choice statement except for the termination condition.

Parallel Blocks

The next element to discuss is the parallel block. Similar to an atomic statement a parallel block specifies a set of responsibilities and statements that can be viewed as a single responsibility enclosed by one or more operators, is essence a parallel block may be viewed as containing one or more atomic statements. These atomic statements may be considered sub-scenarios, or embedded scenarios. The following is a copy of the ACL scenario used for our discussion.

```
Scenario TakeCourses {
   Trigger(observe(TermStarted)),
   parallel {
      Contract Course course = InstanceBind CurrentCourses(iterator);
      Check(CurrentCourses().Contains(course.bindpoint));
      (
         parallel {
           (DoAssignment(course,context))[course.Parameters.NumAssignments];
         }
         |
         parallel {
            (DoMidterm(course,context))[course.Parameters.NumMidterms]
         }
         |
         DoProject(course,context)[course.Parameters.HasProject]
      ) [observe(TermEnded)],
      DoFinal(course,context)[course.Parameters.HasFinal];
      alternative(not observe(LastDayToDrop)) {
         DropCourse(course);
      };
   }[CurrentCourses().Length() | DroppedCourse (course)],
   Terminate(Fire(CoursesCompleted, context));
}
```

Example of Parallel Blocks from the Student Contract

In addition a parallel block denotes a section of a scenario (or atomic elements) that may have multiple instances running simultaneous. The example illustrates a single student taking the courses they are enrolled in. For example, if a student is enrolled in four courses, when execution reaches the first (or outer) parallel block four instances or sub-scenarios, each corresponding to a single course are created for that student. One thing to note, similar to an atomic statement all elements (or sub-scenarios) of the parallel block must be completed before an element outside the block can be executed. The student can then execute and any of the inner parallel blocks for any of the courses they are registered in.

```
@Scenario(parallel)
public class TakeCourses extends Scenario {
   public TakeCourses(Object o, String name) {
      super(o, name);
   }

   @Override
   public void run() {
      Trigger(Observe(TermStarted));
      parallel (CurrentCourses());
      {
         Course tCourse;
       do {
            tCourse = CurrentCourses().get(iterator);
            {
               parallel (tCourse);
               {
                  DoAssignment(tCourse, (Student)_instance);
               }

               parallel (tCourse);
               {
                  DoMidterm(tCourse, (Student)_instance);
               }

               DoProject(tCourse, (Student)_instance);
            }
            Observe(TermEnded);
            DoFinal(tCourse, (Student)_instance);
         } while ( Iterator() < CurrentCourses().size()
              ||   Observe(DroppedCourse(tCourse)));
         }
         Terminate(Fire(CoursesCompleted,getContext()));
      }
}
```

Mapping a Parallel Statement to Java Code

A couple of issues arose while examining the grammar of the *TakeCourses* scenario illustrated below. First, the *OR* statement denoted by the | character indicates that the elements in the statement may be executed in any order. There is however some ambiguity in how ACL handles the *OR* statement. By traditional standards for the *OR* statement only one element of the *OR* statement has to occur before execution continues. However, in the ACL specifications the claim is that all elements still execute and the *OR* statement simply denotes their execution may occur in any order. It is arguable that the *OR* statement may be executed as a loop, iterating until all elements have satisfied their constraints, however this is not compliant with the current ACL syntax [83], and since for our thesis we wish to adhere as close as possible to the original ACL syntax we did not pursue this option further. For the purposes of this thesis it will be treated as defined in the ACL specifications.

Second, by definition (as previously mentioned) an atomic elements cannot be nested, nor can they contain any parallel statements. As written, in the original grammar the parallel statements are embedded directly in an atomic statement. This requires one of two solutions, both the structure and grammar of the atomic statement has to be completely modified to allow for embedded atomic statements and parallel blocks, or we modify the parallel statement to take on similar functionality and constraints of an atomic statement.

Since parallel statements are already considered to be sub-scenarios, and in addition each element must be completed before any other element outside the parallel block can be executed, parallel statement have the same constraints as an atomic statement. This allows us to treat each parallel block (or sub-scenario) as an atomic statement. With this reasoning we removed the atomic statement from the scenario. An additional item is that the original grammar contains an alternative block that allows the path the scenario to alter if the student drops the course. This presented some

conflicting logic; the ACL specification require that responsibilities within the atomic block be executed and completed as a single unit and in the order specified by the grammar for the atomic element to be considered valid. This would require the element identified in the embedded parallel block be completed prior to the completion of the atomic statement. In effect the alternative statement may not be reachable.

The second issue that arose is that the alternative paths in a parallel block require threads to monitor them and determine if and when a student chooses to drop a course. This may be considered an exit condition for the outer parallel statement. If a student drops a course, the internal parallel statements and elements require no further execution and can terminate. This allows us to change the internal alternative statement to become an additional exit condition. By revising the *DropCourse* to become a new observability *DroppedCourse*, we move the requirement for the alternative statement to the termination condition for the parallel block. To manage the new ACL syntax we must allow for multiple termination condition and related predicate logic.

The code snippet below illustrates the addition of the new observability *DroppedCourse* in the student contract and the corresponding Java code in the student's contract class.

```
Observability Boolean DroppedCourse(tCourse course)
{
   Observe (not LastDayToDrop);
   value = instance.DroppedCourse (course);
}

@Observability(observer)
public Boolean DroppedCourse(Course tCourse) {
   Observe (LastDayToDrop, false);
   return CurrentCourses().contains(tCourse);
}
```

The following ACL snippet illustrates the revised *TakeCourses* scenario with the proposed ACL syntax. However, we must explain the functionality behind the embedded parallel statements. When execution reaches the first embedded parallel block, a sub-scenario and thread is created for

each assignment listed in the course contracts *NumAssignments* parameter value. When the execution reaches the second parallel block a sub-scenario and thread is created for each midterm listed in the course contracts *NumMidterms* parameter value. If the course has a project option another sub-scenario and thread is spawned to manage the corresponding responsibility. All threads are added to the outer parallel blocks thread and are managed as cohesive group.

```
Scenario TakeCourses
{
   Trigger(observe(TermStarted)),
   parallel
   {
      Contract Course course = InstanceBind CurrentCourses(iterator);
      Check(CurrentCourses().Contains(course.bindpoint));
      (
         parallel
         {
            DoAssignment(course,context)[course.Parameters.NumAssignments];
         }
         |
         parallel
         {
            DoMidterm(course,context)[course.Parameters.NumMidterms];
         }
         |
         DoProject(course,context)[course.Parameters.HasProject];
       ) [observe(TermEnded)],
        DoFinal(course,context)[course.Parameters.HasFinal];
   }[CurrentCourses().Length() | DroppedCourse (course)],
   Terminate(Fire(CoursesCompleted, context));
}
```

Revised Take Course Scenario

The inner block is ended by identifying a termination condition *TermEnded*. Upon observation of the termination condition the sub-scenarios are shut down and the final responsibility in the grammar is invoked – *DoFinal***.** The scenario terminates and the *CoursesCompleted* event is fired

In summary, this section has addressed two critical elements in our thesis, 1) atomic statements and 2) parallel blocks. In addition we introduced a modification to the current ACL semantics for the choice element. The first, atomic statements may be considered sub-scenarios and a thread is spawned and managed by the contract monitor. The atomic thread is assigned to the enclosing

scenarios thread group and the temporal grammar and constraints are monitored by the contract monitor. There is only one thread spawned per atomic statement executed.

The second, parallel blocks required an alteration to the original ACL grammar. As we outlined, the constraints placed on parallel blocks are similar in nature to atomic blocks and as such may be considered as a block of atomic statements that run simultaneously. Each atomic statement spawns a corresponding thread and is managed by the contract monitor. Again, these threads are assigned to the enclosing scenarios thread group and managed as a cohesive unit. Another important point identified is that atomic statements cannot be embedded nor can they contain parallel statements. This constraint requires that we removed the atomic statement from the scenarios syntax and modify the alternative statement to be a termination condition. In order to accommodate this functionality the ACL syntax has to be modified to allow parallel blocks repetition statement to contain an exit condition, or in effect have one or more repetition conditions that may contain logical operators.

We also introduced the Cancel observability. The proposed Cancel observability is a built in observability in the VF that allows for the exit or termination of both an atomic statement and a parallel block.

Finally we offered a modification the ACL choice statement that allows for complex choice statements to be identified. We proposed that the default value for the choice statement be set to true and the alternative statement would automatically be invoked if the choice statement evaluates to false.

4.5 Pre, Post Conditions, Checks and Invariants

The ACL pre and post conditions and check condition are implemented using Java assertions. An assertion is a statement in Java that enables you to test your assumptions about your program.

Each assertion contains a boolean expression that you believe will be true when the assertion executes. The use of Java Asserts allows us to design and develop specialized ACL check conditions, error messages and configure the VF and the runtime handling of check conditions. Using Java Asserts allows us to configure the VF in one of three ways, 1) *pass through*, log when a check condition is violated but continue running the validation process (default), 2) *interrupt*, log the check condition and prompt the user whether to continue the validation process or shut it down, and 3) *no tracing*, we can configure the VF to ignore or turn off check conditioning. The functionality required to handle the Pre, Post, Check and Invariant checks are built into the ACLContract abstract class that all contract classes inherit from. The following example of an Invariant is similar in nature to pre and post conditions and use Java asserts in the same way.

```
Invariant IsFullCheck
{
   Check(Students().Length() <= CapSize());
}
```

Example of an ACL Invariant

An invariant is similar in nature to the Pre, Post and Check conditions.

```
@Invariant
public Boolean IsFullCheck()
{
   return Invariant(Students().size() <= CapSize());
}
```

Mapping an Invariant to Java Source Code

The first thing to notice is the statement in the Invariant evaluates to either true or false and is passed through as a parameter to the Invariant method defined in the ACL/VF. The method defined in the ACL/VF executes the assert statement. As previously noted, we can control the sequence of events that occurs in the check conditions. This is identified in the configuration file and the flag is configured when the contract is initialized. Note that the pre and post conditions and check statements will implement the *checkAsserts* method.

```
protected Boolean Invariant (boolean b)
{
   return checkAsserts(b);
}
```

ACL/VF Invariant

```
private boolean checkAsserts (boolean b)
{
   Scanner in = new Scanner(System.in);

   switch (enforceAsserts)
   {
      case default:
           assert b : logMessage("Assertion Checked");
           return true;
      case interrupt:
           if (!b) {
              System.out.println("A check condition failed");
              String s = in.nextLine();
              logMessage(s);
           }
           return true;
      case notrace:
           return true;
   }
}
```

Assertion Handling in the ACL/VF

It should be mentioned that the assert statements used by the pre and post conditions, check statements and invariants can easily be extended to implement ACL belief statements. The belief statement can be passed in or set as the message executed by the assert statement.

4.6 Contract Evaluation Reports (CER)

Contract monitoring and traceability of scenario's and event handling is a critical to runtime validation. Text based log files can quickly grow in size and there is no readily available method to verify the runtime conditions other than manually examining them. This can become time consuming and prone to human error. For instance, when we run multiple instances of the contract classes each executing different paths on a set of scenarios the results logged to the log files become inter twined and dispersed, even when the instances of the contracts classes and related scenarios have unique ID's associated with them. For this reason we developed the framework to

generate a set of HTML reports that include the firing and observation of events, list of courses, students and professors associate with each instance of the IUT grouped by term. An addition report generated details the bindings between monitor stations, contract monitors and the IUT instance they monitor and control.

During runtime validation the station manager invokes the CER and generates a set of xml files that reflects the state of the contract classes after the defined event or set of events have occurred. This is an attribute that can be configured in the configuration. Once the runtime validation process is completed, using the xml files a set of HTML reports are generated. The generation of the HTML is not an automatic process built into the VF and must be invoked by a separate command. Samples of the HTM reports may be found in the Appendices.

The following example illustrates how the monitor station runs a separate thread and waits for the specified event to be fired.

```
protected void run ()
{
   do
   {
      {
         Observe (LastDayToDrop);
         logReports();
       }

      {
         Observe (TermCompleted);
         CheckBreakPoint();
      }
   }
   while ( !done );
   Shutdown(false);
}
```

In the above example a time thread is initialized and waits for the LastDayToDrop event is fired. Once the event has been fired and observed a set of xml files are generated by the CER. If the TermCompleted event is observed it checks to see if any of the exit or termination conditions have been met, and if so logs the current state of the contract classes and shuts down the CER. As

mentioned previously the observe methods are considered secondary threads and are assigned to the primary thread group that controls thread scheduling and their position in the thread queue. The wait time between thread switching threads (or thread yielding) may be manually set in the configuration file using the *tpWaitTime* parameter. Should the monitor station shutdown without observing (the Blackboard) either or both of the two events a corresponding report is sent to the CER. Generation of the HTML is handled by a separate process that is manually invoked.

The following segment of code illustrates how the instance of the IUT is bound to the contract monitor and the instance of the IUT may be accessed directly form the station manager during runtime. Since there are no responsibilities defined in the university contract that generates the required xml files for the CER the IUT save methods are invoked directly by the station manager.

```
protected synchronized void logReports ()
{
   University u =
               (University) getContractMonitor().getContext().getInstance();

   u.saveCourses();
   u.saveStudents();
   u.saveProfessors();

   u.saveAll();
}
```

4.7 Discussion of Methodology and Difficulties

The following section will present the difficulties and major issues that needed to be addressed during the development of the proposed Java based ACL/VF. This will be done through a discussion of the issues addressed and the major changes made through each iteration. Over the course of the case study the project went through eight iterations, or development stages, where each iteration addressed a specific concern or design issue.

Iteration One

The first iteration, or development stage focused primarily on two issues. The first requirement was to determine the development environment that would guarantee platform independents and an industry standard language that provided enough functionality and support that we could develop a validation framework independent of any external or third party tools. For this we choose Java 1.8 as the development platform, XML/JAXB for data persistence and eclipse for our IDE.

The second issue during this iteration addressed the creation of a contract instances for runtime monitoring. For this we focused primarily on the university contract. Creating an instance of an ACL contract requires mapping the core elements of an ACL contract, i.e. responsibilities, observabilities, contract parameters and scenarios to a corresponding Java class. Since there currently is no ACL to Java mapping tool developing these contract classes is a manual process and we aimed to maintain as much of the original ACL's original syntax as possible. The issue was mapping the ACL type to a corresponding Java type. Since the identified ACL elements may take on complex structures and grammar and handle different return types we decided to develop a set of annotations that would identify the ACL type the Java methods are mapped to. We maintain a one to one mapping between the ACL element identified in the contract and a corresponding method in the Java class, including element name, parameter list and return type. We used JUnit to map the ACL check pre and post check conditions to the Java code. This allowed us to build the basic framework to map ACL contracts to Java based contract class. A set of test classes with static data were developed to test the basic functionality of the contract classes.

Iteration Two

The second iteration focused on determining the best course of action to build the validation framework and instantiating the Java contract classes. Borrowing from previous research,

specifically JavaMOP we identified three principal components. The first was the main component responsible for driving the validation framework i.e. startup, configuration, monitoring and shutdown of the validation framework itself – the Station Manager. The second was the concept of a contract monitor. However for this iteration it was an empty class used as a place holder. Other than identifying it was required to manage an instance of an ACL contract no further functionality was identified. We then considered the complex issue of initializing and monitoring multiple instances of multiple scenarios running simultaneously. This required another set of components that would manage multiple instances of a contract or set of contracts running simultaneously. To handle this complex issue we introduced the Monitor Station. Each monitor station was responsible for initializing one or more instance of a contract monitor and ensuring each instance of the contract monitors was independent and their context was safe from interruption from other contract monitors.

The main issue that arose during this iteration was the instantiation process and the management of the contract instance in the Monitor Station. We addressed this issue by developing a container that would store the separate instance of the contract monitors and use the hashcode generated by the JVM when an object is instantiated. We also used JAXB to write an XML file that would persist the mapping and the current state between the Monitor Stations and the Contract Managers it was responsible for managing.

Iteration Three

This iteration focused on the initialization and management of a contract instance and its related scenarios. The primary issues was developing a method that would monitor and log the temporal grammar of the scenarios. This was a complex issue since no previous research was conducted on developing a framework that could reflect the temporal grammar in real time. For

this we borrowed from Buhr's and Casselman's work on Use Case Maps. In their book they describe scenarios as time threads. This provided the principle solution to the issue faced. In order to develop a system that is "time sensitive" a mechanism was required that could simulate the passage of time or temporal grammar during runtime validation. We developed the notion of an ACL clock and created a configuration file that would allow us to set the start and end time for the simulation and the frequency and duration the clock changes time. However, we needed to determine how the validation framework would interact with the clock. We decided to borrow from the Observer pattern and have the station managers register with the clock and be notified of any changes in the state of the clock. The station managers then pushes the change to the contract monitors. This required that each contract manager register with their associated station manager.

The next issue was the initialization or instantiation of the contract class or set of classes and binding them to the contract monitor. To address this issue we looked at the structure of the ACL contracts and utilized the syntax and semantics that denotes the relationship between contracts. The university contract is identified as the main contract, this indicates the university contract class needs to be instantiated before any other contract is initialized. However, since the contracts and contract scenarios are inter dependent the initialization sequence is critical. Since the main contract is the university contract we have the station manager instantiate the station monitors on startup. In order to control the number of station managers to start up we added two other attributes to the configuration file that would identify the number of station managers to initialize and the number of contract monitors to assign to each station manager. However, we still needed to initialize and bind the main contract to the contract monitor. This required the implementation of Java reflection. The required number of IUT instances to run simultaneously is identified in the configuration file. On startup station manager initializes the required number of IUT instances, and passes them to

the monitor station or stations. The distribution of IUT instances to monitor station is dependent on the number of contract monitors assigned to each monitor station. For this iteration we chose a one to one relationship. Implementing reflection the station manager creates an instance of a contract monitor and binds the IUT instance to the contract monitor. Through reflection the contract monitor class is generated and initialized, the name of the contract monitor reflects the class name of the IUT object passed in. In order to monitor the runtime and log the initialization process we implemented Apache's Log4j.

Iteration Four

Iteration four focused on initializing the contract scenarios, managing and monitoring the temporal grammar and the monitoring of event firing and observation. We first noticed that scenarios come in one of two forms. The first form was what we later deemed a sub-scenario. These scenarios were time sensitive, i.e. have no temporal grammar but are triggered by the observation of an event. The second, and more complex scenario is time sensitive and has to adhere to temporal grammar. In addition, in their grammar they contain sub-scenarios that are defined elsewhere in the contract that are dependent upon the observation of an event. To address this issue we implemented Java threads and introduced the notion of a new ACL type of scenario – the TimeThread. An ACL TimeThread has temporal grammar defined by its grammar and is triggered and terminated by a sequence of events that are separated by a distinct and identifiable period of time - *linear space*. In addition the issue of identifying the event fired with correct instance of the IUT became critical.

This required that all scenarios identified as a TimeThread inherit from a TimeThread class. The TimeThread class is responsible for starting up and managing the threads and thread groups required to maintain the temporal grammar identified in the grammar. In addition it required the

development of a set of observability methods that would "wait" for the required event to fire and return the new state of the IUT. For this we modified the Blackboard class developed in previous work. The only addition to the Blackboard class was the requirement to identify which contact manager was responsible for firing the current event and when. For this we developed a container that allows the Blackboard to group the events fired with the contract monitor and the related instance of the IUT that fired them. This allowed us to ensure that the events pushed to the scenarios is associated with the correct instance of the IUT and contracts.

Iteration Five

Iteration five focused primarily on testing the framework and scenario monitoring and managing the temporal grammar. For this we faced two issues, the first was the development of valid test data, and the second was the ability to dynamically alter a scenarios path.

We required test data for classes, students and professors. In addition we needed to develop a method to test the temporal grammar of scenarios and thread management. For the first three requirements we developed a set of xml files that contained data that reflect real world conditions as close as possible. We created a set of DAO (Data Access Objects) that read and write to these files and creates a corresponding set of files that store the actual data (courses, students and professors) created and the instance of the IUT they are assigned to. In order to simulate temporal grammar we created a set of eight default terms.

In order to develop a runtime environment where the data related to the individual instances of the IUT are different we created a set of Helper classes for the contracts. These helper classes randomly select the data and the amount of from the corresponding data files and assign them to the IUT during the initialization process. The limitation of the data elements is determined by a set of attributes defined in the configuration file. Each attribute has a min and max range. However

another consideration arose, the temporal grammar and the notion that the runtime validation process may continue over several terms. To address this issue we have the validation frame work clean up the data when a term ends and generate new data before the next term starts.

However, in order to monitor the temporal grammar of scenarios and develop alternative paths for each instance of the scenarios we devised a method to take the default term information and randomly generate the start date, end date, registration opened (which is before the term starts), registration closed, which is within fourteen days of the start of the term. The generated dates are within a range of seven days (plus or minus) from the default dates defined in the terms data file.

Iteration Six

Iteration six focused on the course contract and developing the ACL check conditions and inter contract communication.

In the previous iterations the ACL check conditions were implemented using JUnit, specifically the assertTrue assertion. However, when running our test cases if an assertion failed our test case would terminate. In effect it would not pass through. In the previous iteration we developed a method, which has and attribute in the configuration file that allowed us to turn off the check conditions during runtime validation. This was not a viable long term solution. We looked into developing our own specialized asserts that would reflect our needs more accurately. For this we implemented Java Asserts. Along with providing a basic set of predefine asserts that manage both pre-conditions and post-conditions (i.e., Pre, Post, Check and Invariant), Java Asserts allows us to define our own asserts and the functionality associated with these asserts. In addition we can define and configure our own runtime functionality to implement at runtime, *i.e. pass-through, interrupt and no-trace.*

The next issue was inter contract communication or the correct identification of the events themselves and the messages they contain. For this we extended the previous version of the ACL event, extended it and developed the concept of an event handler. All ACL events now extend the abstract ACLEvent class. The Event Handler allows for the dynamic creation of events as they are fired and identifies the time they are created, the station monitor, contract monitor, instance of the contract and the IUT class that fired them. In effect we now manage event creation through a central agent that creates specialized ACL events. This allows for the identification and development of specialized events. Through reflection and the Event Handler any method observing an event can determine the events current state, the contract monitor and IUT instance that fired it. In addition when the event is observed additional information is passed to it, i.e. the time it was observed and the contract instance that observed it.

Iteration Seven

Iteration seven primarily focused on the student contract and the more complex issues of the atomic statement and parallel blocks. Our methodology and logic behind our design decisions was discussed in detail in the previous section, but we did not discuss what led to these design issues and results.

First we considered each atomic element and identified blocks of code in the parallel blocks that represented similar types of atomic elements. As defined in the ACL specifications atomic elements may be looked at as sub-scenarios. The grammar and structure of these atomic elements require thread monitoring and thread control. This quickly led to an explosion of threads being generated and resulted in the runtime validation process thrashing the operating system, eventually resulting in a complete system crash. In order to control this we needed to limit the amount of

threads that were running simultaneously to one thread per atomic statement and one thread to handle all the parallel blocks.

When reviewing the grammar of the parallel blocks we noticed that the assignments block only needs to be active in order determine if an assignment has been completed, similar reasoning applied to the midterms. The project thread, if required, only had to be active to determine the assignment of a team, determine if the team has been assigned and to work on the project. This led to the solution, first – assignments have a due date. The assignments sub-scenario only has to be invoked when the due date is reached to check to see if an *AssignmentComplete* event has occurred. However the specialized event has to contain the course id, assignment number, due date and current status of the assignment (complete | due). The same logic applied to the midterms and project. This meant that if the main thread was notified of the due dates then the threads would only have to be invoked at these specific times in the temporal grammar. The main thread could invoke the required thread (in its group) and yield control to the thread on an "*on and when*" required basis. Once the sub-scenario was completed thread control was passed back to the main thread and the sub-scenario would "*sleep*" again; ergo only one thread was required to be active at any one time in order to manage the temporal grammar and flow.

Iteration Eight

Throughout the previous iterations monitoring and tracing was primarily logged to a text file using Apache's Log4j.These log files soon grew large and were hard to dissect and discern specific information or determine the temporal grammar of a scenario was being met. For example, all instances of an IUT, or multiple IUT's were logged to the same log file so the results were dispersed and intertwined. In addition, even though the events are logged to an xml file (one to

one) there soon became an unmanageable amount of trace files and there was no discernible way to determine which file equated to which event.

In order to develop a more comprehensive and human readable method to determine the outcome we decided to develop the framework to generate a set of reports. For this we created as set of classes that are responsible for creating a set of html reports. The reports generated are 1) monitors, lists the station managers, contract monitors and associated IUT.s including the time of creation and destruction by term for each instance of the IUT, 2) a list of students and the courses the student is registered in for each IUT - grouped by term, 3) list of courses for each IUT and the professor assigned to it – grouped by term, 4) list of professors for each IUT and the courses assigned to the professor – grouped by term, 5) list of terms and there related dates – grouped by university and 6) two event reports, the first reports on the observable event types, and the second reports on the non-observable event types. Samples of these reports may be found in Appendix A.

4.8 Conclusion

From the above case study it has been shown that ACL scenarios, atomic elements and parallel blocks contain many intricacies that need to be addressed when translating ACL contracts to a set of Java based contract classes.

The method was to develop 1) a base for mapping an ACL contract to a Java class representation that matched the ACL syntax as close as possible that the VF could instantiate and monitor, 2) develop a VF classes that would provide the functionality required to provide the runtime validation in accordance the ACL specifications, 3) develop a monitor based solution that provided the functionality to monitor multiple instances of a set of scenarios running simultaneously and 4) the preliminary design and implementation of a set of reports used for validation.

During the design and development phases logs were generated that used to verify at runtime the functionality of the ACL/VF. Throughout the iterations the messages sent to log file were improved and contained additional information that allowed us to determine the proper operation of the functionality being introduce. These log files were examined at the end of every runtime verification process to determine if the IU calls matched the required sequence calls – i.e. honored the temporal grammar. Incorrect sequences and values were also implanted in the IUT to determine that indeed errors were caught by the monitors and correctly logged. Later in iteration eight we developed the framework for creating traceable reports. Examples of these traceability matrices may be found in the Appendices. Examples of the log files are available on this thesis's website.

Once it was verified that the solution presented was indeed a feasible solution we went back to the ACL contracts and performed an element by element mapping from ACL to the proposed Java contract classes and determined the feasibility and structure in the Java based contract classes and any future requirements that may need to be implemented in the proposed ACL/VF. The results of this study are covered in the next section.

5. Mapping ACL to Java Classes

The following chapter will provide a summary of the mapping from ACL to Java. There will be no description of the ACL elements and their syntax, however they can be found in the ACL/VF specifications [11]. As such we limit our discussion to describing how each element may be implemented in the Java environment. Any mapping not mentioned in this section has not been implemented and as such future research into these mapping must be considered.

5.1 Using Declarations

The current model of the monitor stations and contract monitors is developed using the Java languages framework and as such allows for the direct mapping of import and export statements.

5.2 Contracts

Contracts are encapsulated in a contract monitor, in effect each contract has a corresponding contract monitor that handles all pre, post and invariant checks. It also has one or more specifications for scenarios contained within the contract. For example, as detailed in Chapter Four each scenario that contains either atomic statements or parallel block (or both) will have one monitor thread running for the scenario and one thread running for each atomic statement or block structure. A block structure contains specific grammar and defines a sequence of events that must be executed in a specified sequence, and as such may be considered atomic elements in their own rights. In addition each contract monitor contains an instance of the contract class in the form of a Java class. The Java classes contain all the contract scope variables, parameters, observabilities, responsibilities and scenarios. In addition the station managers and contract managers inherit from a set of abstract classes that encapsulates the functionality to run and monitor events and temporal sequence of the scenarios.

Since the contracts are Java based classes, modeling and developing inheritance based contracts as defined in the ACL specifications is fully supported.

5.3 Bind Point Expressions

IUT type instance.bindpoint

Since binding the IUT instance is done during the initialization phase of the contract monitor, the contract instance corresponding to the IUT type instance can be retrieved directly form the contract monitors instance after initialization. In addition since the contract instance is bound directly to its monitor station the instance of the IUT is available directly from the instance of the monitor station.

contract class instance.bindpoint

Upon creation each contract class has a local variable that stores a reference to the IUT instance the IUT instance is bound to. This variable is accessible from any monitor station or contract monitor that are bound together through the monitor station. In effect, the IUT instance may be considered a global variable that has contract monitor scope.

5.4 Context Access Expression

A variable is included in each contract monitor (called context) that provides access to context or contract instance. This variable stores a reference to the instance of the contract that the IUT object is bound to. In addition there is an instance variable declared in each contract that allows access to the IUT bound to the contract instance.

5.5 Value Access Expression

Accessing return values in the Java based classes has a direct one to one mapping from either a responsibility or an observability. Handling return statements and accessing values is handled using Java getters and setters methods. Any information that is required to be passed through to

inter-dependent contracts can be retrieved by calling the contracts instance directly and invoking the required method similar in nature to querying an instance of a class in the same scope as the calling class in a Java program.

5.6 Parameter Access Expression

Since parameters are included in the contract classes, the contract class reference variables may be called directly form the contract monitors instance of the contract. Since the contract class is bound to the monitor station the contract class can be accessed by other contracts that are in the same scope as the monitor station.

5.7 Don't Care Expression

Currently the *dontcare* expression is limited to the String data type and is representative of passing the null data type as a parameter. However, this method does not work for other data types such as an Integer. Further research is required into developing a global *dontcare* expression that can be representative of any ACL type and corresponding Java data type.

5.8 Variables

Java supports all the variable types listed in the ACL specifications. The only consideration to take into account is the scope of the variables in the ACL contract. If it is a contract scope variable it is included in the corresponding contracts Java class. If it is a variable from a responsibility, it is also included in the corresponding contracts Java class. If it is a variable from a responsibility where its usage spans two or more events, it is included in the contract monitors scope that contains the instance of the contract. For example, a variable declared in a scenario is stored in the parent class of the scenario (TimeThread) class which is bound to the corresponding contract monitor.

5.9 Parameters

Parameters are handled in a similar fashion as variables. They are included in the Java contact class that corresponds to the ACL contact they are declared in. Parameter values may be declared final, static. Their values cannot be modified by an external class or contract monitor that has access to its context. All parameters are bound to the contact during the initialization process. This requires that the contract class creates a set of access methods if the value of the parameter is allowed to be modified by external contract classes or the contract monitor.

5.10 Structure

By implementing Java assertions and asserts into the framework we can now support static checks. In addition we can configure the VF handle the static checks during runtime in one of three ways; *pass-through, interrupt and no trace*. However, the ACL structure grammar was not addressed in our thesis.

5.11 Observability Method

Bound Observability

Bound observabilities are handled in the contracts Java class that corresponds to the contract the observability is defined in. They are defined as Java methods that uses the contracts class instance variable to make a call to the necessary IUT method to retrieve the required information.

We identify this method as an observability by adding a new Java annotation @Observability(bound). Note that the assertion parameter identifies the type of observability we are mapping to.

Unbound Observability

Defined, or unbound observabilities are also included in the corresponding contracts Java class and mapped directly to Java methods. There is a direct one-to-one mapping between the content defined observabilities in the ACL contract and in methods in the corresponding contract classes.

We identify this method as an observability and the type by adding a new Java annotation @Observability(unbound).

Observe

This is new type of observability we introduced in our thesis. These observabilities are also included in the contracts Java class and Java methods. However, these observabilities require and event to be fired before it proceeds to execute the requirements set out in the contracts associated grammar. The functionality behind these observabilities is encapsulated in the VF framework and is not required to be reflected in the corresponding Java code. This allows us to maintain a one to one mapping between the contracts observability and the corresponding method in the contracts Java class without adding the overhead of adding additional code to manage the observation of an event.

We identify this method as an observability and the type by adding a new Java annotation @Observability(observe).

5.12 Invariants

Like observabilities, invariants are included as methods in the contact classes. Each invariant method is developed using Java Assert statements to run the checks defined in the ACL contract they originate from. This allows us to develop custom asserts that can control the flow of events and define specific error messages. This extends the use of invariant checks in the contract

monitors and allows the contract monitors to 1) make the checks, 2) control the flow of the validation process, 3) capture error message and 4) send the messages to the CER.

5.13 Responsibilities

Bound Responsibilities

Contract responsibilities map directly from a contract to the corresponding Java classes. All check conditions are performed by the containing contract monitor as discussed in the above section on invariants. The execute statement in bound to the method in the IUT bound to the contract and accessible through the instance variable.

Bound responsibilities are identified by the annotation @Responsibility (bound);

Unbound Responsibilities

Each unbound responsibility contains a grammar of responsibilities and maps directly to a Java method in the contracts Java class. An unbound responsibility may also include the execute statement and as such performs the execute statement as defined in the bound responsibility method.

Unbound responsibilities are identified by the annotation @Responsibility(unbound);

Special Responsibilities

New

Each instance of the contracts Java class has a New method defined that maps directly to the definition in the corresponding contract. The New method is invoked one of two ways, first when an instance of the contracts Java class is created, and the second when a New event is fired and defined in the contract as a triggering event that is required to start a responsibility. This is similar to the new event based observability defined earlier. New responsibilities are identified by the annotation @Responsibility (new);

Finalize

As in the New responsibility each instance of the contracts Java class has a Finalize method defined that maps directly to the definition in the corresponding contract. The Finalize method is invoked by the contract manager when an instance of the contracts Java class is destroyed. Finalize responsibilities are identified by the annotation @Responsibility (finalize);

5.14 Responsibility Bodies

Responsibility bodies are mapped directly to the corresponding contracts Java class that contains the responsibility. The list of contract instances are maintained by the contract monitor and may be used to retrieve the contract instances by any contract that is bound to the enclosing monitor station.

Scalar type variables:

If no contract instance exists, execution will terminate with an exception. Termination and logging of the current state of the contract instance is managed by the monitor station

List variables

Are handled the same as Scalar type variables

Belief Statement

Currently not supported

Pre Statement

Pre statements are converted directly to the corresponding Java Assert statement developed for our solution

PreSet Statement

Currently not supported. Currently we manually perform the PreSet in the responsibility body of the corresponding contracts Java class

Post Statement

Currently handled the same way as Pre Statement and has a one to one mapping to the Java Assert statement developed for our solution

Fire Statement

The fire statement is implemented in the VF and fires the event to the Blackboard. A call to the Blackboards fire statement is embedded in the VF code that handles firing an event. Additional information is added to the event before it is fired to the Blackboard, including the time the event was fired, the context the event was fired in, the contract monitor and monitor station the contract is bound to, and any other information that may be relevant to the firing of the event. Once fired to the Backboard, it is available to all contract monitors to observe.

Execute

The execute statement is bound to the corresponding method in the contracts Java class. The Java method invokes the corresponding method call in the IUT with the instance variable bound to the contracts Java class.

5.15 Stub Responsibilities

Stub responsibilities are not implemented in our proposal. Original research into stubs indicated they could be handled as sub-scenarios. However further research into this area is required.

5.16 Scenarios

For our thesis we have identified two primary types of scenarios, a main scenario and a sub-scenario. Main scenarios are considered to be either event driven or time sensitive. Event driven scenarios have to honor a specific sequence of events and responsibilities identified by the scenarios grammar. Time sensitive scenarios are driven by events that are time sensitive, i.e. are fired at specified times intervals during the life of the scenario. The time sensitive events in the scenario are separated by a defined time interval. For example the Term scenario has special events, i.e. Term Started, Term Ended that are dependent upon the date-time element configured for the specific term. A sub-scenario is not time sensitive but may require the observation of an event before it can proceed. For example a sub-scenario may require that the MarksRecorded event be fired before it can proceed to start the scenario.

Event and time sensitive scenarios have their own Java thread associated with them and are managed by the enclosing contract monitor. Sub-scenarios spawn a secondary thread and is added to the main scenarios thread group. As the validation process iterates through the main scenario control is passed to the current observability (sub-scenario) or responsibility identified in the scenario. The main thread passes control to the secondary thread and waits until the secondary thread returns control.

Since scenarios contain observabilities and responsibilities there is a one to one mapping between a contract scenario and the corresponding TimeThread class (inner class) in the Java code. In addition there is one to one between the scenarios grammar and the scenarios body in the contracts Java code. Scenario elements are identified with the @Scenario (TimeThread) and @Scenario (stub) annotations.

Trigger Statement

A trigger statement for a responsibility is handled by the new event observability method implemented in our solution. The event observability method is passed through to the trigger as a parameter. Upon successful observation of the defined event the trigger is fired and control is passed to the scenario's thread.

Terminate Statement

In our proposal there is one of two ways to handle the terminate statement. The first is the inclusion of an event that must be observed. If the event is observed the scenario is terminated, if not the scenario loops around itself and repeats the process. A second method is to define a termination condition (this type of termination process we identify as an exit condition) that must be met. These condition are identified in the configuration file. The conditions are set up in the CheckBreakPoint method provided in the VF. If the exit conditions are met the scenario is terminated and the system is shut down. This type of termination condition is used to shut down the scenarios and related contract monitors in a single process; basically this is considered a force shutdown.

Atomic Element

Atomic elements are considered sub-scenarios and as such have their own Java thread assigned to them. The grammar of an atomic element has a one to one mapping with the contacts Java code. When an atomic element is invoked it spawns a thread and the thread is assigned to the main scenarios thread group managed by the contract monitor. When an atomic element is reached in the main scenarios grammar, control is passed over to the atomic elements thread and the main scenarios thread waits (sleeps) until control is returned. This enforces that only those events within

the atomic element can occur and only in the specific sequence identified by the grammar is complete. Atomic elements are identified with the @Atomic annotation

Parallel Block

As previously discussed, the block elements of a parallel block may be viewed as individual atomic elements embedded with the parallel block. These block elements spawn a thread for each block and are assigned to the main scenarios thread group, again this is managed by the contract monitor. This ensures that each block element acts like and adheres to the atomic elements constraints. Remember from the previous discussion presented in Chapter Three, these atomic elements pass control between each other and only one thread is active at a time. This is critical in controlling combinatorial explosion. Parallel elements are identified with the @Parallel annotation

Alternate Element

Alternate elements comes in one of two forms, an alternative to a choice statement or as an alternative to an atomic statement. In the first form the alternative statements grammar in the contract maps directly to a corresponding method in the contracts Java code. Only one thread is spawned by a choice statement, either from the choice block or the alternative block. The same logic applies for an atomic statements alternative statement.

The New Instance

The new instance was not implemented in our solution and as such requires further research

Observe Element

The observe element is handled in the same fashion as the event observability element defined earlier. The observe element is identified with the @Observe annotation

One or More (+) Repeat Operator

These operators indicate that we are dealing with a scenario that must be invoked at least once and may loop back on itself and repeat the specified grammar until an exit condition is met. These operators map to an exit condition that must be identified by the CheckBreakPoint method in the VF and the attributes managed in the configuration file. Since there is currently no tool to dynamically assign these conditions, we must enter a set of parameters (exit conditions) manually in the configuration file can be passed as a variable to the VF. Currently it is restricted to one parameter, the breakpoint parameter in the configuration file.

Zero or More (*) Repeat Operator

This operator works in the same manner as the previous repeat operator with the exception the trigger statement may never occur. If the trigger statements requirement is not fired before the break point (or exit condition) is reached a message is sent to the CER indicating the scenario was never triggered.

6. Conclusion and Future Work

In this section we will present a brief discussion of our work and our conclusion. We will close our thesis with a brief discussion of future work that can be directed from our research.

6.1 Discussion

Our work in this thesis illustrates the feasibility of mapping the ACL specification language to the Java framework. We have illustrated the mapping through a complex case study and identified how the various elements of the ACL specifications language and semantics can be mapped to the Java framework. In the university system case study we illustrated how ACL contracts can be mapped directly to their corresponding Java classes and support the contract variables of ACL. These variables can be dynamic in nature and can be updated throughout the execution of an IUT. As such they can be used to manually drive test cases to alter a scenarios path during runtime verification. We also illustrated how we can map responsibilities (both bound and unbound), and observabilities (both bound and unbound) to equivalent methods in Java.

In addition, this case study illustrates how to map some of the more complex and intricate elements of an ACL contract onto their related Java code. We focused mainly on ACL scenarios and the various temporal semantic elements they support, namely: atomic elements, parallel blocks, alternatives, and various combination of these. We also showed how parallel blocks may be considered blocks of atomic statements and as such may be considered a set of sub-scenarios that maps directly to a set of methods or blocks of code in the corresponding Java contract classes.

In order to support the ACL/VF functional requirements and monitor multiple scenarios running simultaneously we implemented a set of ACL specific monitors. Each possible contract instance has a corresponding contract monitor that controls the lifespan of the contract. Monitors include variables and events that are used to drive test cases as well as support the runtime

verification of scenarios. A monitor station contains the instances of the IUT and the related contract instance they are bound to. In addition the monitor station contains the global variables that are shared amongst contracts and is responsible for event handling. The monitor station manages inter-contract and inter-scenario communication through the Blackboard.

We have suggested that ACL scenarios may be of one of three types 1) event sensitive, in that they must honor the sequence of events and responsibilities identified in their grammar, 2) time sensitive, in that the events and responsibilities of such a scenario are separated by a defined period of time and the events to be observed are fired at specific intervals identified in the temporal grammar, and 3) sub-scenarios, scenarios that are embedded within a main scenario. We have illustrated how each of these three types of scenarios can be mapped directly to corresponding Java code. In particular, in order to address the second type of scenario we introduced our own background process, the ACL clock. The Blackboard and all monitor stations are registered with the clock. The clock allows us to develop test cases that fire events at specific times and/or time intervals and monitor the temporal ordering of events.

In summary, we have shown that ACL contract specifications and their corresponding runtime verification mechanisms can be supported in the Java framework by implementing a validation framework that 1) implements a monitoring system and 2) has a set of Java classes that are mapped directly from the ACL specifications that use the monitoring system to implement the functional requirements for the ACL/VF. Additionally, we have discussed at length how to map elements of an ACL contract to corresponding Java code.

6.2 Conclusion

The two current versions of the ACL/VF have a number of issues that prevent them from being usable tools for scenario testing. In particular, the original version is .NET specific and

incorporates external tools that are no longer supported. Unfortunately upgrading it to a more recent version of .NET would amount to a complete rewrite. Another version of the ACL/VF has been implemented and tested with the JavaMOP framework. This approach requires manually mapping an ACL specification onto JavaMOP monitor specifications. Furthermore, it cannot provide the functionality required to monitor multiple scenarios running simultaneously.

In order to avoid these problems, we propose re-implementing the scenario monitoring aspects of ACL/VF through a mapping to an industry standard language, namely Java. In this thesis we have successfully accomplished this mapping for a set of contracts (which are part of a large case study) developed independently of our work. This reimplementation of ACL/VF through a mapping to the Java framework is not only feasible (as we have demonstrated) but likely constitutes the basis for a full implementation of ACL/VF using Java.

6.3 Future Work

Given we have demonstrated the feasibility of a mapping from ACL to Java, we now briefly identify additional feature and refinements that can further improve this functionality.

6.3.1 ACL Specifications

In our thesis we identified the ACL specifications that were relevant to scenario monitoring and specific to our case study. However, future work is required to address the complete mapping of the ACL/VF specifications to Java.

6.3.2 ACL to Java Parsing Tool

As mentioned above, there is no tool to generate the proposed Java contracts classes from their corresponding ACL contract specifications. Future work should look into developing a parsing engine that would automatically generate the contracts Java source code from the ACL specifications and create the bindings to the IUT.

6.3.3 Testing Platform

Currently there is no method to input a set of test cases and manually validate the runtime execution. Future work should look into developing a front end tool that would allow the creation of manual test plans.

6.3.3 Contract Evaluation Reports

Currently we generate a set of contract reports from xml files. A front end tool could be developed that would allow users to query the xml files and develop a set of specialized reports (which, in turn, could improve how traceability is addressed). An alternative is develop a database that is ACL specific and use a generic reporting tool, for example Crystal Reports

6.3.4 Common Monitoring Model

The Common Monitoring Model of Java is both an information model specifying monitored objects and a data model specifying uniform values such as the operational status values. As part of the information model, CMM also defines the attributes of an object, for example the number of requests handled by a service, and relations between objects, such as the fact that a service is hosted on a certain computer. Future work should investigate the relevance (if not possible reuse) of CMM in supporting ACL/VF.

6.3.5 Use Case Maps

Use case maps are a scenario-based modeling technique for describing causal relationships between responsibilities of one or more use cases. Use case maps allow for the ability to model dynamic systems that may alter paths at run-time. In addition use case maps are intuitive and relatively simple to learn.

Currently ACL contracts are textual artefacts created manually. Semantically, as previously mentioned, they are close to UCMs (with respect to their use of responsibilities and scenarios).

But ACL contracts are significantly more detailed (i.e., closer to code) than UCMs. The integration of these two notations (and of their corresponding tools) seems highly interesting as it could enable the design of a graphical front end tool based on UCMs and capable of generating ACL contracts (which in turn could generate Java code).

6.3.6 Distributed ACL

In Chapter Three we briefly introduced the idea that by implementing a middle layer that would bind the Responsibilities and Observabilities to a set of Java Connector Classes, or implement Java's Service Oriented Architecture the proposed solution can be bound to any IUT independent of the underlying software environment. This can be the foundation for future research into developing a distributed ACL/VF system.

7. Bibliography

[1] Rational Software White Paper, "Rational Unified Process: Best Practices for Software development Teams," Rational Software, Cupertino, CA, 1998.

[2] W. W. Royce, "Managing the Development of Large Software Systems: Concepts," The Institute of Electrical and Electronics Engineers, 1970.

[3] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer,* vol. 21, no. 5, pp. 61-72, May 1988.

[4] "Guide to the Software Engineering Body of Knowledge," 2014.

[5] R. S. Aguilar-Saven, "Business process modelling: Review and framework," vol. 90, no. 2, pp. 129 - 149, 2004.

[6] H. Saiediana, "Requirements engineering: making the connection between the software," Information and Software Technology 42 (2000) 419–428, 1999.

[7] M. Gorschek, "Requirements engineering: In search of the dependent variables," Information and Software Technology 50 (2008) 67–75, 2007.

[8] B. J. Shena Hui, "Integration of business modelling methods for enterprise information system analysis and user requirements gathering," Computers in Industry 54 (2004) 307–323, 2003.

[9] B. Meyer, "The Unspoken Revolution in Software Engineering," IEEE Computer, Vol 39, no 1, pp 121 - 123, ETH Zurich, 2006.

[10] M. Glinz, "On Non-Functional Requirements," in *Requirements Engineering Conference,15th IEEE International*, Delhi, 2007.

[11] D. Arnold, "An Open Framework for the Specification and Execution of a Testable Requirements Model," Ottawa, Ontario, 2009.

[12] R. Saiediana H. and Daleb, "Requirements engineering: making the connection between the software developer and customer," vol. 42, no. 6, pp. 419 - 428, 2000.

[13] C. Auruma, "The fundamental nature of requirements engineering activities as a decision-making process," Information and Software Technology Vol 45 pp 945–954, 2003.

[14] D. H. S. Janzen, "On the Influence of Test-Driven Development on Software Design," Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS, 2006.

[15] K. Beck, JUnit Pocket Guide, Cambridge, MA: O'Reilly Media, 2004.

[16] D. J. Nidhra Srinivas, "Black Box and White Box Testing Techiques - a Literature review," International Journal of Embedded Systems and Applications (IJESA) Vol.2, No.2, 2012.

[17] J. Dooley, "Unit Testing in Software Development and Professional Practice," *Apress,* pp. 193 - 208, 2011.

[18] T. Cheon, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," 2003.

[19] B. Turk Dan, "2014," Journal of Database Management, Volume 16, No. 4, pp. 62-87 , Assumptions Underlying Agile Software Development Processes.

[20] B. Turk Dan, "Limitations of Agile Software Processes," Processes in Software Engineering,pp. 43-46, Alghero, Italy, 2002.

[21] C. Forward, "Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals," in *2008 international workshop on Models in software engineering*, New York, NY, 2008.

[22] A. Bertolino, "Software Testing Research: Achievements, Challenges and Dreams. In proceedings of the Future of Software Engineering," in *FOSE '07 2007 Future of Software Engineering*, Minneapolis, MN, 2007.

[23] J. Cocks, "Model Driven Design," *Insight,* vol. 7, no. 2, pp. 5-8, 2015.

[24] J. Beltramin, "Mapping ACL to JavaMOP: A Feasibility Study," Ottawa, Ontario, 2014.

[25] R. Binder, Testing Object-Oriented Systems, Boston, MA: Addison-Wesly Professional, 2000.

[26] A. Gokhale, "Developing applications using model-driven design environments," Computer, Vol 39 , No 2, 2006.

[27] G. Sivanageswara, "Rational unified process for service oriented application in extreme programming," Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference, Tiruchengode, India , 2013.

[28] R. Soley, "Model Driven Architecture," Object Management Group, White Paper , 2000.

[29] L. Apfelbaum and J. Doyle, "Model Based Testing," *Software Quality Week Conference,* 1997.

[30] P. A., "Model-based testing," Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference. pp. 15-21, St. Louis, Missouri, 2005.

[31] J. Tuma, "Methods of Automated Model Transformations in Information System Analysis," Procedia Technology Vol. 8 pp 612 – 617 , 2013.

[32] R. Clark, "Object-Oriented Theories for Model Driven Architecture," in *OOIS 2002 Workshops, LNCS 2426*, Montpellier, France, 2002.

[33] S. J. Mellor., "Model Driven Architecture," J.-M. Bruel and Z. Bellahsène (Eds.): OOIS 2002 Workshops, LNCS 2426, pp. 290-297, 2002.

[34] "Introduction To OMG's Unified Modeling Language," Object Management Group, 2015.

[35] L. Y. Briand, "A UML-Based Approach to System Testing," A UML-Based Approach to System Testing, Ottawa, 2001.

[36] P. Aksit, "Use Cases in Object-Oriented Software Development," AMIDST, 1999.

[37] I. Jacobson, Object Orientated Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[38] R. J., "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Tests," University of Zurich, Institute for Informatics, 2000.

[39] H. P. Weidenkaupt, "Scenario Usage in System Development: A Report on Current Practice.," In proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice, 1998.

[40] I. Ul Haq, "Modeling Use-Case Sequential Dependenicies using ACL," Carleton University, Department of Computer Science, Ottawa, Ontario, 2014.

[41] M. A. Wirfs-Brock, "Object Design: Roles, Responsibilities and Collaborations," Addison-Wesley Professional, 2002.

[42] M. Ryser, "Using Dependency Charts to Improve Scenario-Based Testing: Management of Inter-Scenario Relationships," In proceedings of the 17th International Conference on Testing Computer Software, TCS2000, pp. 1-10, Washington, DC, 2000.

[43] P. Weidenkaupt, "Scenario Usage in System Development: A Report on Current Practice," In proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice pp. 222-241, 1998.

[44] B. R., "Testing Object Oriented Systems: Models Patterns and Tools," Addison-Wesley, 2000.

[45] R. J. A. C. R. Buhr, Use Case Maps for Object-Oriented Systems, Prentise Hall, 1996, pp. 7, 144.

[46] H. Xiao-ling D., "An Approach to Event-Driven Software Testing," Tianjin, China, 2002.

[47] W.M.P., "Formalization and verification of event-driven process chains," Information and Software Technology, Vol 41, No. 10, pp. 639–650, 1999.

[48] N. P. Etzion O., "Event Processing in Action," Manning Publications, 2010.

[49] G. Verbeek Mendling, "Detection and prediction of errors in EPCs of the SAP reference model," vol. 58, no. 6, p. 578–601, 2008.

[50] D. Meyer, "Applying Desing by Contract," Computer (IEEE), 1992.

[51] J-P Corriveau, W. Shit and D. Arnold,, "Reconciling Offshore Outsourcing with Model-Based Testing," *Software Engineering Approaches for Offshore and Outsourced Development,* pp. 6-22, 2010.

[52] J-P Corriveau, W. Shit and D. Arnold,, "Modeling and Validating Requirements using Executable Contracts and Scenarios," *Proceedings of Software Engineering Research, Management & Applications,* pp. 311-320, 2010.

[53] J-P Corriveau, W. Shit and D. Arnold, "Scenario Based-Validation: Beyond the User Requirements Notation," in *Software Engineering Conference (ASWEC)*, Auckland, New Zealand, 2010.

[54] D. Arnold, "An Open Framework for the Specification of Execution of a Testable Requirments," 2009.

[55] "JavaMOP Versions," 2015. [Online]. Available: http://fsl.cs.illinois.edu/index.php/All_JavaMOP_Versions. [Accessed 9 Aug 2015].

[56] C. Nebut, "Automatic Test Generation: A Use Case Driven Approach," IEEE Transactions on Software Engineering, Vol. 32, No. 3, 2006, 2006.

[57] W. Schiller, "Case Studies and Tools for Contract Specifications," University of Washington, Seattle, WA, 2014.

[58] "Use Cases based Requirements Validation with Scenarios," Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, Ottawa, ON, 2005.

[59] R. S. Maiden N., "Developing use cases and scenarios in the requirements process," ICSE '05 Proceedings of the 27th international conference on Software engineering, New York, NY.

[60] D. Arnold, "Another Contracting Language 3.2 Specification Document," 2010.

[61] D. Arnold, "The University: Contract Evaluation Framework Walk Through 2.1," 2008.

[62] W. Grieskamp, "Multi-paradigmatic model-base testing," *FATES/RV,* vol. 4262, pp. 1 - 19, 2006.

[63] L. B. Utting, "A taxonomy of model-based testing.," University of Waikato, 2006.

[64] E. Stolz, "Temporal assertions using AspectJ," vol. 144, pp. 109-124, May 2006.

[65] K. D'Amorim, "Jeagle: a JAVA Runtime Verification tool," pp. 1-20, July 2013.

[66] D. Drusinsky, "The temporal rover and the ATG rover," vol. 1885, K. P. J. a. V. W. Havelund, Ed., Springer, 2000, pp. 323 - 330.

[67] D. Farago, "Model-based Testing in Agile Software Development," 2010.

[68] A. Broy, Eds., Model-Based Testing of Reactive Systems: Advanced Lectures, vol. 8, Springer, pp. 69-86.

[69] H. Holzmann, "A Practical Method for Verifying Event-Driven Software".

[70] A. M. Memon, "Developing Testing Techniques for Event-driven Pervasive," College Park, MD 20742, 2004.

[71] J-P Corrieveau, W. Shi and D. Arnold, "A Scenario Driven Approach to Model-Based testing".

[72] P. O. J. a. W. W. Metz, "Against Use Case Interleaving," Springer-Verlag, Ireland, 2001.

[73] D. Kealey, "Towards the Automated Conversion of Natural-Language use Cases to Graphical use Case Maps," *Electrical and Computer Engineering,* pp. 2377 - 2380, May 2006.

[74] D. Amyot, "Use Case Maps: Qucik Tutorial," University of Ottawa, Ottawa, ON, 1999.

[75] D. Amyot, "Use Case Maps as a Feature Description Notation," *FIREworks Feature Constructs Workshop,* pp. 27-44, May 2000.

[76] R. Andrade, "Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks," Ottawa, ON, 2000.

[77] T. Alsumait, "Use Case Maps: A Visual Notation for Scenario-Based User Requirements," *10'th International Conference on Human-Computer Interaction,* June 2003.

[78] J. Hassine, "Early modeling and validation of timed system requirments using Timed Use Case Maps," vol. 20, no. 2, pp. 181-211, June 2015.

[79] J. Hassine, "Timed Use Case Maps," in *System Analysis and Modeling: Language Profiles*, vol. 4320, Springer, 2006, pp. 99-114.

[80] M. Miga, "Deriving Message Sequence Charts from Use Case Maps Scenario Specifications," *Lecture Notes in Computer Science ,* vol. 2078, pp. 268-287, June 2001.

[81] D. Amyot, "User Requirements Notation: The First Ten Years, The Next Ten Years," *Lournal of Software,* vol. 6, no. 5, May 2011.

[82] D. Arnold, "The University: Contract Evaluation Framework Walkthrough Version 2.1," Ottawa, ON, 2008.

[83] D. Arnold, "Another Contracting Language 3.2: Specification Document," 2010.

[84] D. Arnold, J.-P. Corriveau and W. Shi, "Modeling and Validating Requirements using Executable Contracts and Scenarios," in *Proceedings of Software Engineering Research, Management & Applications*, Montreal, Canada, 2010b.

[85] "IEEE Standard for System and Software Verification and Validation," *IEEEE Std 1012-2012 (Revision of IEEE Std 1012-2004),* pp. 1-223, May 2012.

[86] M. Leucker, "Teaching Runtime Verification," *Runtime Verification,* pp. 34-48, 2012.

[87] G. J. Bollella, "The Real Time Specification for Java," Computer (Volume:33 , Issue: 6), 2000.

[88] G. Hilderink, "Comminicating Java Threads," 20th World Occam and Transputer User Group Technical Meeting, WoTUG-20, Enschede The Netherlands, 1997.

[89] M. W. McBeth, "Executing Java threads in parallel in a distributed-memory environment," CASCON '98 Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, 1998.

[90] "J2EE Connector Architecture White Paper: Integrating Java applications with existing Enterprise Applications," Oracle Technology Network.

[91] Farrell, W., "Introduction to the J2EE Connector Architecture," 2002.

[92] J. C. C. N. D. Aldrich, "Archjava: Connecting Software Architecture to Implementation," University of Washington : Department of Computer Science and Engineering, Orlando, FL, 2002.

[93] D. Panda, "An Introduction to Service-Oriented Architecture from a Java Developer Perspective," O'Reilly Online, 26 01 2005. [Online]. Available: http://archive.oreilly.com/pub/a/onjava/2005/01/26/soa-intro.html. [Accessed 10 Aug 2015].

[94] D. Arnold, "The University: Contract Evaluation Framework Walkthrough," Ottawa, ON, 2008.

[95] D. Arnold, "ACL Contract Language 3.2: Specifications Document," Ottaw, ON, 2010.

[96] P. J. Winkler Stefan, "A survey of traceability in requirements engineering and model-driven development," Journal of Software and Systems Modeling , Vol 9 (4), pp 529-565, Secaucus, NJ, 2010.

[97] N. P. Etzion O., "Event Processing in Action," Manning Publications Co., Greenwich, CT., 2010.

[98] G. Luo, "RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties," University of Illinois at Urbana-Champaign, Urbana IL, 2014.

[99] P. N., "Aspect-Oriented Programming: An Introduction to Aspect-Oriented Programming and AspectJ," Department of Technology, University of Kalmar, Kalmar, Sweden, 2002.

[100] A. D. W. M. Mussbacher G, "Visualizing Early Aspects with Use Case Maps," Transactions on AOSD III, LNCS 4620, pp. 105–143, Berlin, Heidelberg, 2007.

[101] A. Weiss J., "Functional Testing of Complex Event Processing Applications: Requirements for running functional tests against CEP applications," Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on, 2001.

[102] C. Y. Leavens G.T., "Design by Contract with JML," 2006.

[103] H. U. B. J. Karaorman M., "jContractor : A Reflective Java Library to Support Design By Contract," Lecture Notes in Computer Science Volume 1616, 1999.

[104] M. H., "Using Assertions in Java Technology," Oracle Technologies.

[105] F. S. Laboratory, "JavaMOP Agent," University of Illinois at Urbana-Champaign: Department of Computer Science , Urbana-Champaign, IL, 2014.

[106] G. McCluskey, "Using Java Reflection," Oracle Tecnology Network, 1998.

[107] T. J. Tutorials, "About the Java Technology," Oracle Technology Network, 2015.

[108] "Programming With Assertions," [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html. [Accessed 9 Aug 2015].

[109] "Sun Java Enterprise System 5 Monitoring Guide," 2015. [Online]. Available: http://docs.oracle.com/cd/E19528-01/819-5081/6n76iq52h/index.html. [Accessed 10 Aug 2015].

[110] R. G. Legunsen O., "Evolution-Aware Monitoring-Oriented Programming," University of Illinois at Urbana-Champaign, 2015.

[111] J. Beltramin, "Mapping ACL to JavaMOP: A Feasibility Study," Ottawa, Ontario, 2014.

[112] L. Y. Briand, "A UML-based Approach to System Testing," *Journal of Software and Systems Modeling,* vol. 1, no. 1, 2002.

[113] A. C. J.-P. a. S. W. Arnold, "Modelling and Validating Requirements Using Executable Contracts and Scenarios," *Software Engineering Research and Applications (SERA),* pp. 311-320, May 2010.

[114] D. Arnold, "An Open Framework for the Specification and Execution of a Testable Requirements Model," Ottawa, ON, 2009.

Appendices

Copies of the source code illustrated in this thesis, the ACL contracts, log4j trace files and copies of the reports may be found at the following URL. Also note there are copies of the data files that the test data is generated from and a copy of the xml files generated during runtime data. The complete trace files are located in the logs directory. There are also copies of the properties files that can be viewed.

https://drive.google.com/folderview?id=0B5cT_BjwahZrQWpiTl9iVXpFYW8&usp=sharing

Appendix A: Review of Verification of our Solution

For testing purposes we developed a configuration file that allowed us to define the number of instances of the IUT and the set of associated contracts to test concurrently. Each instance of the IUT and its set of contracts with have different values for their attributes, e.g. number of students, number of courses, passing grade etc., in effect creating variations for the same scenario or set of scenarios defined in the contracts. Recall that an instance of the IUT may have multiple instances of associated contracts, so multiple instances of the IUT will have multiple instances of the set of contracts and associated scenarios that are defined in the contracts.

The Station Manager contains a collection of monitor stations that are initialized at runtime and managed throughout the lifespan of the ACL test case. Once the monitor station are initialized we then register an instance or set of instances of the IUT with the monitor station. Recall that the number of monitor stations and the number of instances of the IUT to register with the monitor station are configured in the configuration file.

The following section of code details the logic we follow in setting up and configuring the ACL Validation Framework. First we need to initialize the newly defined ACL Clock. We then proceed to validate that all our required parameters are set as defined in the configuration file through the trace files, the xml test cases generated and the start and stop times of the ACL clock.

We then configure the station manager and instantiate the number of IUT instances to test. Each instance of the IUT is registered with the Station Manager. We then configure and setup the monitor stations; in our example one monitor station per instance of the IUT. A background thread is started, starts up the monitor stations and waits for the termination condition to be met. In this case we set a start date and end date that is monitored by the ACL Clock. Once the termination condition is met (the end date) the station manager shutdowns all instances of the monitor stations,

contract monitors and cleans up the instances of the IUT. The Station Manager is then shut down and generates a set of reports (See Appendices for examples of the reports).

```
public void startup()
{
   startClock();
   checkSetupParameters();
   configureStationManager();
   setupMonitorStations();

   try
   {
      Thread t = new Thread (_instance);
      t.start();

      // We need to wait a while and allow the Clock and Blackboard to
      // synchronize before starting up the background processes and
      // monitor stations

      Thread.sleep(ConfigManager.getInstance().waitTime());
                startupMonitors();
      } catch (InterruptedException e)
      {
      // perform a clean shutdown and log the shutdown process
         shutdown(true);
      }
      bp.start();
}
```

Another important aspect of the Station Manager to mention is that it is responsible for starting and stopping a set of background processes that are used by the validation framework; for example the ACL clock and instances of test cases that determine when and which professor and/or student to terminate. The background processes have the same lifespan as the station manager, i.e. the lifespan of the ACL validation process.

In order to test our proposal we setup a configuration file that allows us to assign the ranges defined in the ACLContract. For testing purposes we developed a random number generator that returns a random number in the range defined in the ACL contracts and configures the contracts accordingly.

The following is an example of the elements of the configuration file that we are concerned with. As you will notice randomly create 8 to 12 courses and related contracts, 5 to 8 students and related contracts and 12 to 18 professors. Professors have no related contract.

```
#IF set to 0 [default 0] 1:1 ratio between Monitors and Contracts
numOfMonitors=0
numOfUniversities=3    // Number of main contracts to initialize
#For Testing Purposes
#Range [min | max]
courses=[8..12]
students=[5..8]
professors=[12..18]

# [3 | default 6]
maxCoursesForFTStudents=5
maxCoursesForPTStudents=2
passRate=70
```

Generating the Test Data

A university contains multiple instances of courses, students and professors. In our test cases a course is randomly assigned a professor and a random number of students up to the limit constrained by class size. The maximum class size is dependent upon a configuration variable set in the configuration file - more on this later. A professor may be assigned to one, two or the maximum of three classes, or not may not be assigned to a class. The number of professors assigned to a university is limited and dependent upon another variable set in the configuration file. A student may register in one to a maximum number of courses that is dependent on their student status (i.e. full-time or part-time). In addition a university has a number of terms it can iterate through, again this is a configurable parameter. As the VF iterates through each term, the scenarios are responsible for creating and adding courses, creating and adding students and creating and adding professors to the instance of the IUT. In addition the VF assigns professors to the courses, and allows students to register or drop courses during the specific time frame

associated with each term. In addition the VF monitors student assignments, midterms, projects, final exams and registration of the final grades for all students.

Validation Reports

One a runtime was completed, we generated a set of reports. We first examine the Terms reports (Appendix B1) and ensure that the required number of terms were generated and each term had a random dates assigned to them.

We then examined the monitor report (Appendix B2). In this report we verified that a monitor station was generated for each instance of the university contract and a corresponding set of contract monitors were created for the courses and students. You will notice that we were able to test each monitor type separately, in this case we tested the student's monitors. You will also note that we are able to track the student the contract monitor was generated for.

We then proceed to verity that the courses, students and professors were generated according to our defined test cases, and ensure that each student was registered in the required number of courses (according to their status) and that a professor was assigned to each course.

The critical report, the events report was examined last. In this report we are able to track that all events defined in the scenarios are fired and observed and in the sequence defined in the temporal grammar.

Summary

In summary, according to both the logs and reports generated our proposal met all our requirements.

Appendix B1: University Term Report

UID	OID	Term	Semester	Year	Start Date	End Date	Last Day to Regiseter	Last Day to Drop
355629945	355629945	1	WINTER	2015	Jan/05/2015	Apr/22/2015	Apr/13/2015	Jan/16/2015
355629945	355629945	2	SUMMER	2015	May/08/2015	Jun/20/2015	Jun/09/2015	May/21/2015
355629945	355629945	3	SUMMER	2015	Jul/01/2015	Aug/16/2015	Aug/04/2015	Jul/12/2015
355629945	355629945	4	FALL	2015	Sep/08/2015	Dec/25/2015	Dec/18/2015	Sep/21/2015
355629945	355629945	5	WINTER	2016	Jan/08/2016	Apr/18/2016	Apr/09/2016	Jan/15/2016
355629945	355629945	6	SUMMER	2016	May/07/2016	Jun/18/2016	Jun/09/2016	May/15/2016
355629945	355629945	7	SUMMER	2016	Jul/01/2016	Aug/22/2016	Aug/10/2016	Jul/12/2016
355629945	355629945	8	FALL	2016	Sep/03/2016	Dec/20/2016	Dec/12/2016	Sep/14/2016
824909230	824909230	1	WINTER	2015	Jan/07/2015	Apr/22/2015	Apr/14/2015	Jan/18/2015
824909230	824909230	2	SUMMER	2015	May/04/2015	Jun/17/2015	Jun/04/2015	May/14/2015
824909230	824909230	3	SUMMER	2015	Jul/02/2015	Aug/18/2015	Aug/06/2015	Jul/11/2015
824909230	824909230	4	FALL	2015	Sep/01/2015	Dec/26/2015	Dec/17/2015	Sep/09/2015
824909230	824909230	5	WINTER	2016	Jan/03/2016	Apr/16/2016	Apr/04/2016	Jan/13/2016
824909230	824909230	6	SUMMER	2016	May/03/2016	Jun/18/2016	Jun/06/2016	May/11/2016
824909230	824909230	7	SUMMER	2016	Jul/05/2016	Aug/19/2016	Aug/10/2016	Jul/17/2016
824909230	824909230	8	FALL	2016	Sep/04/2016	Dec/20/2016	Dec/09/2016	Sep/17/2016
1347137144	1347137144	1	WINTER	2015	Jan/05/2015	Apr/21/2015	Apr/09/2015	Jan/17/2015
1347137144	1347137144	2	SUMMER	2015	May/03/2015	Jun/13/2015	Jun/02/2015	May/15/2015
1347137144	1347137144	3	SUMMER	2015	Jul/05/2015	Aug/13/2015	Aug/01/2015	Jul/14/2015
1347137144	1347137144	4	FALL	2015	Sep/03/2015	Dec/22/2015	Dec/14/2015	Sep/10/2015
1347137144	1347137144	5	WINTER	2016	Jan/11/2016	Apr/14/2016	Apr/06/2016	Jan/24/2016
1347137144	1347137144	6	SUMMER	2016	May/05/2016	Jun/14/2016	Jun/04/2016	May/17/2016
1347137144	1347137144	7	SUMMER	2016	Jul/04/2016	Aug/15/2016	Aug/07/2016	Jul/12/2016
1347137144	1347137144	8	FALL	2016	Sep/04/2016	Dec/22/2016	Dec/12/2016	Sep/12/2016

Appendix B2: Monitor Report

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	1	2015	WINTER	Jan/07/2015	Apr/20/2015	Jan/14/2015	Apr/10/2015

Monitor Station: 1588970020 Number Of Monitors: 8

Monitor Id	Monitor Type	Instance id	Type	Thread Id	Type
712025048	UniversityContractMonitor	824909230	University	24	TimeThread[University.Term]
712025048	UniversityContractMonitor	824909230	University	24	TimeThread[University.Term]
712025048	UniversityContractMonitor	824909230	University	24	TimeThread[University.Term]
712025048	UniversityContractMonitor	693412362	Student	25	Scenario[Student.[Mohamed, Sano].RegisterForCourses]
712025048	UniversityContractMonitor	601449476	Student	28	Scenario[Student.[Benton, Morse].RegisterForCourses]
712025048	UniversityContractMonitor	1103172529	Student	30	Scenario[Student.[Robin, Anselmo].RegisterForCourses]
712025048	UniversityContractMonitor	1771604075	Student	32	Scenario[Student.[Elmer, Boykin].RegisterForCourses]
712025048	UniversityContractMonitor	1708143520	Student	34	Scenario[Student.[Ted, Houghton].RegisterForCourses]

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
355629945	1	2015	WINTER	Jan/05/2015	Apr/19/2015	Jan/16/2015	Apr/08/2015

Monitor Station: 770189387 Number Of Monitors: 8

Monitor Id	Monitor Type	Instance id	Type	Thread Id	Type
315138752	UniversityContractMonitor	355629945	University	23	TimeThread[University.Term]
315138752	UniversityContractMonitor	355629945	University	23	TimeThread[University.Term]
315138752	UniversityContractMonitor	355629945	University	23	TimeThread[University.Term]
315138752	UniversityContractMonitor	925255477	Student	26	Scenario[Student.[Bernardo, Guerra].RegisterForCourses]
315138752	UniversityContractMonitor	2017721442	Student	27	Scenario[Student.[See, Saidi].RegisterForCourses]
315138752	UniversityContractMonitor	771487565	Student	29	Scenario[Student.[Mohamed, Sano].RegisterForCourses]
315138752	UniversityContractMonitor	2077610438	Student	31	Scenario[Student.[Benton, Morse].RegisterForCourses]
315138752	UniversityContractMonitor	2127507046	Student	33	Scenario[Student.[Robin, Anselmo].RegisterForCourses]

Appendix B3: University Course Report

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	1	2015	WINTER	Jan/07/2015	Jan/07/2015	Apr/14/2015	Jan/18/2015

UID	Course Code	Title	Department	Professor	Capacity	Assignments	Midterms	Final	Project	Students
772777427	COMP 1805	Discrete Structures I	Computer Science	Bertossi,Leopoldo	65	3	0	true	false	3
1921595561	COMP 2804	Discrete Structures II	Computer Science	Carmichael,Gail	46	1	0	true	true	1
739498517	ANTH 3027	Studies in Globalization and Human Rights	Sociology-Anthropology	Shotwell,Alexis	87	1	1	true	true	0
314265080	ANTH 4007	Advanced Studies in Anthropological Theory and Methods	Sociology-Anthropology	Shepherd,John	65	1	0	true	true	1
1627800613	ANTH 4590	Capstone Seminar in Globalization, Culture, and Power	Sociology-Anthropology	Shepherd,John	93	3	0	true	false	3
1919892312	SOCI 3570	Studies in Art, Culture and Society	Sociology-Anthropology	Kennelly,Jacqueline	97	4	0	true	false	4
114935352	SOCI 4020	Advanced Studies in Race and Ethnicity	Sociology-Anthropology	Kennelly,Jacqueline	97	3	0	false	false	3
515132998	SOCI 4043	Advanced Studies in the Sociology of the Family	Sociology-Anthropology	Shotwell,Alexis	68	1	1	true	true	3
359023572	SOCI 4650	Advanced Studies in Power and Everyday Life	Sociology-Anthropology	Kennelly,Jacqueline	62	2	1	false	false	1
1130478920	SOCI 4750	Advanced Studies in Globalization and Citizenship	Sociology-Anthropology	Shotwell,Alexis	107	1	1	false	true	0
971848845	SOCI 4910	Tutorial in Sociology	Sociology-Anthropology	Caputo,Tullio	38	1	1	true	true	3

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	2	2015	SUMMER	May/07/2015	Jul/22/2015	Aug/06/2015	Jul/11/2015

UID	Course Code	Title	Department	Professor	Capacity	Assignments	Midterms	Final	Project	Students
2058534881	PSYC 3702	Perception	Psychology	Parlow,Shelley	72	2	1	true	false	0
305623748	ANTH 1001	Introduction to Anthropology	Sociology-Anthropology	Stasiulis,Daiva	110	3	0	true	false	1
758529971	ANTH 1002	Introduction to Issues in Anthropology	Sociology-Anthropology	Vallée,Michel	105	3	1	false	false	2
1192108080	ANTH 2510	Theories of Human Nature	Sociology-Anthropology	Chen,Xiaobei	71	2	1	true	false	0
1227229563	SOCI 4820	Field Placement: Research and Analysis	Sociology-Anthropology	Chen,Xiaobei	45	4	0	true	false	1
1982791261	SOCI 4830	Advanced Studies in Applied Social Research	Sociology-Anthropology	Chen,Xiaobei	60	1	0	true	true	4
1562557367	SOCI 4850	Contemporary Problems in Sociology	Sociology-Anthropology	Cummings,June	117	1	1	true	true	1
1101288798	SOCI 4860	Contemporary Problems in Sociology	Sociology-Anthropology	Vallée,Michel	42	4	1	true	false	1
942731712	SOCI 4900	Honours Thesis	Sociology-Anthropology	Vallée,Michel	40	2	0	false	false	1
971848845	SOCI 4910	Tutorial in Sociology	Sociology-Anthropology	Caputo,Tullio	38	1	1	true	true	3
1910163204	SOCI 4920	Tutorial in Sociology	Sociology-Anthropology	Cummings,June	110	1	1	true	true	2

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	3	2015	FALL	Sep/04/2015	Dec/18/2015	Dec/17/2015	Sep/09/2015

UID	Course Code	Title	Department	Professor	Capacity	Assignments	Midterms	Final	Project	Students
1051754451	COMP 4001	Distributed Computing	Computer Science	Lanthier,Mark	47	3	1	true	false	0
1349277854	COMP 4002	Real-Time 3D Game Engines	Computer Science	Morin,Pat	47	4	1	true	false	0
305808283	SOCI 4701	Special Topic in Criminal Justice and Social Policy	Sociology-Anthropology	Pollard,George	94	2	1	true	false	0
2111991224	SOCI 4702	Special Topic in Criminal Justice and Social Policy	Sociology-Anthropology	Clement,Wallace	82	4	1	true	false	0
292938459	SOCI 4703	Special Topic in Criminal Justice and Social Policy	Sociology-Anthropology	Clement,Wallace	84	4	1	true	false	0
917142466	SOCI 4710	Directed Research in Power and Everyday Life	Sociology-Anthropology	Clement,Wallace	84	3	0	false	false	0
1993134103	SOCI 4720	Research Placement in Power and Everyday Life	Sociology-Anthropology	Pollard,George	79	4	1	true	false	0
405662939	SOCI 4730	Colonialism and Post-Colonialism	Sociology-Anthropology	Kelly,Katharine	62	2	1	false	false	0
653305407	SOCI 4740	Advanced Studies in Subjectivity	Sociology-Anthropology	Pollard,George	69	2	1	true	false	0
1404928347	SOCI 4760	Advanced Studies in Time and Space	Sociology-Anthropology	Davies,Darryl	60	3	1	true	false	0
604107971	SOCI 4770	Advanced Studies in Governmentality	Sociology-Anthropology	Kelly,Katharine	60	2	1	true	false	0

Appendix B4: University Professors Report

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	1	2015	WINTER	Jan/07/2015	Apr/17/2015	Apr/14/2015	Jan/18/2015

OID	Emp Id	Department	Name	Email Address	Courses		
[redacted]	[redacted]	Computer Science	Carmichael,Gail	gail_catsignscs.carleton.ca	COMP 2804		
		Computer Science	Bertossi,Leopoldo	bertossiatsignscs.carleton.ca	COMP 1805		
		Computer Science	Morin,Pat	morinatsignscs.carleton.ca	COMP 3203	COMP 1601	COMP 4002
		Computer Science	Chiasson,Sonia	chiassonatsignscs.carleton.ca			
		Computer Science	LaLonde,Wilf	lalondeatsignscs.carleton.ca			
		Psychology	Bowker,Anne	anne.bowker@carleton.ca	PSYC 3600		
		Psychology	Lacroix,Guy	guy.lacroix@carleton.ca			
		Psychology	LeFevre,Jo-Anne	jo-anne.lefevre@carleton.ca			
		Psychology	Paquet,Lise	lise.paquet@carleton.ca			
		Psychology	Gick,Mary	mary.gick@carleton.ca			
		Psychology	Serin,Ralph	ralph.serin@carleton.ca			
		Psychology	West,Robert	robert.west@carleton.ca			
		Sociology-Anthropology	Shotwell,Alexis	alexis_shotwell@carleton.ca	ANTH 3027	SOCI 4043	SOCI 4750
		Sociology-Anthropology	Kennelly,Jacqueline	jacqueline_kennelly@carleton.ca	SOCI 4020	SOCI 3570	SOCI 4650
		Sociology-Anthropology	Shepherd,John	john_shepherd@carleton.ca	ANTH 4590	ANTH 4007	ANTH 4915
		Sociology-Anthropology	Tfaily,Rania	rania_tfaily@carleton.ca			
		Sociology-Anthropology	Caputo,Tullio	tullio_caputo@carleton.ca	SOCI 4910		

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	2	2015	SUMMER	May/07/2015	Jul/18/2015	Aug/06/2015	Jul/11/2015

OID	Emp Id	Department	Name	Email Address	Courses		
[redacted]	[redacted]	Computer Science	van Bunny,Bugs	bugsvanbunny@cmail.carleton.ca			
		Computer Science	Deugo,Dwight	deugoatsignscs.carleton.ca			
		Computer Science	Carmichael,Gail	gail_catsignscs.carleton.ca	COMP 2804		
		Computer Science	Corriveau,Jean-Pierre	jeanpieratsignscs.carleton.ca			
		Computer Science	van Kaick,Oliver	oliver.vankaickatsignscs.carleton.ca			
		Computer Science	White,Tony	arpwhiteatsignscs.carleton.ca			
		Computer Science	LaLonde,Wilf	lalondeatsignscs.carleton.ca			
		Psychology	Bennell,Craig	craig.bennell@carleton.ca			
		Psychology	Leth Steensen,Craig	craig.lethsteensen@carleton.ca			
		Psychology	Mantler,Janet	janet.mantler@carleton.ca			
		Psychology	Nunes,Kevin	kevin.nunes@carleton.ca			
		Psychology	Parlow,Shelley	shelley.parlow@carleton.ca	PSYC 3702		
		Sociology-Anthropology	Shotwell,Alexis	alexis_shotwell@carleton.ca	ANTH 3027	SOCI 4043	SOCI 4750
		Sociology-Anthropology	Stasiulis,Daiva	daiva_stasiulis@carleton.ca	ANTH 1001		
		Sociology-Anthropology	Cummings,June	junecummings@cmail.carleton.ca	SOCI 4850	SOCI 4920	
		Sociology-Anthropology	Vallée,Michel	michel_vallee@carleton.ca	SOCI 4860	SOCI 4900	ANTH 1002
		Sociology-Anthropology	Chen,Xiaobei	xiaobei_chen@.carleton.ca	ANTH 2510	SOCI 4830	SOCI 4820

Appendix B5: University Students Report

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	1	2015	WINTER	Jan/07/2015	Apr/17/2015	Apr/14/2015	Jan/18/2015

OID	Student #	Gender	Name	Status	Failures	Courses Taken					
[redacted]	[redacted]	FEMALE	[redacted]	Full Time	0	ANTH 4915	ANTH 4007	ANTH 4805	COMP 2805	SOCI 2550	SOCI2005
		MALE		Full Time	0	ANTH 4007	COMP 1805	ANTH 4915			
		FEMALE		Part Time	0	SOCI 3570	ANTH 4007	ANTH 4805			
		MALE		Full Time	0	SOCI 4043	ANTH 4915	SOCI 3570			
		FEMALE		Part Time	0	SOCI 4910					
		FEMALE		Full Time	0	ANTH 4915	SOCI 4910	COMP 1805	SOCI 4020	SOCI 3570	
		MALE		Full Time	0						

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	2	2015	SUMMER	May/07/2015	Jul/18/2015	Aug/06/2015	Jul/11/2015

OID	Student #	Gender	Name	Status	Failures	Courses Taken					
[redacted]	[redacted]	FEMALE	[redacted]	Part Time	0	SOCI 4830	SOCI 4900	SOCI 4850	ANTH 1002	COMP 1005	COMP 1805
		MALE		Full Time	0	SOCI 4900	SOCI 4830	ANTH 1001	SOCI 4920		
		MALE		Full Time	0	SOCI 4850					
		FEMALE		Full Time	0	SOCI 4830	ANTH 1002				
		MALE		Part Time	0	SOCI 4920					
		FEMALE		Full Time	0	SOCI 4820	ANTH 1002	SOCI 4860	SOCI 4910		
		MALE		Part Time	0	SOCI 4830					

Appendix B6: Events Report

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	1	2015	WINTER	Jan/07/2015	Apr/17/2015	Apr/14/2015	Jan/18/2015

Base Monitor	Event ID	Event Type	Time	Monitor ID	Contract ID	Thread ID	Thread Name	Fired	Observed	Data
1579572132	1438033654284	New	01/02/15	1250391581	739498517	19	UniversityContractMonitor [1250391581]	true	true	new()
1579572132	1438033655017	ProfessorsCreated	01/04/15	1250391581	739498517	26		true	true	
1579572132	1438033656363	ProfessorsAssigned	01/05/15	1250391581	739498517	26		true	false	
1579572132	1438033661788	TermStarted	01/09/15	1250391581	739498517	26		true	true	Year [2015] Term [WINTER] StartDate [Jan/07/2015] EndDate [Apr/22/2015]
1579572132	1438033664056	LastDayToRegister	01/10/15	1250391581	739498517	23		true	false	Year [2015] Term [WINTER] StartDate [Jan/07/2015] EndDate [Apr/17/2015]
1579572132	1438034228690	LastDayToDrop	04/15/15	1250391581	739498517	26		true	true	Year [2015] Term [WINTER] StartDate [Jan/07/2015] EndDate [Apr/17/2015]
1579572132	1438034275567	TermEnded	04/26/15	1250391581	739498517	23		true	true	Year [2015] Term [WINTER] StartDate [Jan/07/2015] EndDate [Apr/17/2015]
1579572132	1438034307076	New	06/23/15	1250391581	739498517	0	Background process	true	true	new()

UID	Term Number	Year	Semester	Start Date	End Date	Last Day to Register	Last Day to Drop
824909230	2	2015	SUMMER	May07/04/2015	Jul/18/2015	Jun/04/2015	May/14/2015

Base Monitor	Event ID	Event Type	Time	Monitor ID	Contract ID	Thread ID	Thread Name	Fired	Observed	Data
1579572132	1438034307842	CoursesCreated	07/18/15	1250391581	739498517	19		true	true	
1579572132	1438034311846	StudentsCreated	04/28/15	1250391581	739498517	19		true	false	
1579572132	1438034316083	ProfessorsCreated	06/25/15	1250391581	739498517	19		true	true	
1579572132	1438034322618	ProfessorsAssigned	05/01/15	1250391581	739498517	19		true	false	
1579572132	1438034330967	TermStarted	07/19/15	1250391581	739498517	19		true	true	Year [2015] Term [SUMMER] StartDate [May/04/2015] EndDate [Jul 18/17/2015]
1579572132	1438034335095	LastDayToRegister	05/05/15	1250391581	739498517	19		true	false	Year [2015] Term [SUMMER] StartDate [May/04/2015] EndDate [Jul/18/2015]
1579572132	1438034602668	LastDayToDrop	06/04/15	1250391581	739498517	19		true	false	Year [2015] Term [SUMMER] StartDate [May/04/2015] EndDate [Jun/17/2015]
1579572132	1438034817046	TermEnded	08/21/15	1250391581	739498517	19		true	true	Year [2015] Term [SUMMER] StartDate [May/04/2015] EndDate [Jul/18/2015]
1579572132	1438034843008	CoursesDestroyed	06/21/15	1250391581	739498517	19		true	false	
1579572132	1438034845058	StudentsDestroyed	06/23/15	1250391581	739498517	19		true	true	
1579572132	1438034847244	ProfessorsDestroyed	06/23/15	1250391581	739498517	19		true	false	