The University

Contract Evaluation Framework Walk Through

Dave Arnold
School of Computer Science
Carleton University
Document Version 2.1
05/23/2008

Contents

Abstract

A university system is used to present an open framework for the specification, execution, and evaluation of scenarios. Our framework supports the binding of a testable model against a candidate implementation for evaluation. Such evaluation includes the invocation of static and dynamic checks, the execution of scenarios, and the capture and evaluation of metrics. Metric evaluation allows our framework to consider both functional and non-functional requirements. The grocery store is used to provide a complete end-to-end walk through of our framework's capabilities and operation. The walk through will begin with the elicitation and specification of scenarios and finish with a contract evaluation report.

Introduction

The following document contains a simple example based on a university registration system, the execution and the completion of a term. The purpose of the example is to illustrate the elicitation of contracts, scenarios, and interactions for the case study. In addition, the example will illustrate the binding and execution tasks performed by the framework. The document is organized as follows. A description of the case study will first be presented. Next, four modular contracts will be derived and their syntax and semantics will be discussed in detail. Our example will then define a series of scenario relations based on the case study. As will be seen shortly, a scenario relation is used to define constraints on the execution of scenarios. Our example will be based on the second version of the ACL and our case study is designed to present the new features found within ACL 2.0.

Once the contracts have been presented and discussed, the contract compilation process will be outlined, followed by the binding steps required to map the elements defined within the contracts to the Implementation Under Test (IUT). Once binding is complete, the static check procedure is presented along with the necessary IUT instrumentation required for the evaluation of scenario execution is discussed. The examination of execution metrics and non-functional requirements is then presented. Finally, the document concludes with ah discussion of the contract evaluation report.

Case Study

The case study used within this example is modeled after a physical university. A university was selected because it requires little domain knowledge from the reader and the university example has also been used as a canonical example in other works. A university can be viewed as a set of students who are part of a university. The university creates a set of courses each and every term. Once course creation is completed, students register in courses, with a maximum number of courses constrained by their student status (full/part-time). Once students have completed their course registration, the term begins. During a term, students must complete their courses by doing assignments, mid-terms and final exams. Some courses also have group projects that also must be completed. Once the term ends, each course reports final grades to the university, and the university decides if a given student is allowed to continue with his/her education. Additional details regarding the case study will be provided as each of the contracts is presented.

The following assumptions will be made in order to simplify the example:

- Courses span a single term.
- Courses have single sections.
- Courses have at least one grade element (i.e. no satisfactory or incomplete courses).

Of course, there are a lot of other aspects within a complete university registration system. However the short description shown above will introduce several scenarios, scenario interactions, and

non-functional requirements.

Step 1 – Contract Creation

Initially, we need to specify the contracts. We will begin by looking at a modular contract design. That is, we will create four separate contracts that define the university system. After the four separate contracts have been specified, we will present a series of relations that define how the scenarios defined within the four contracts interact. The four contracts and the relations would then be assembled into a single contract project. A contract project consists of three elements:

1. One or more contracts and contract relations.
2. An IUT to evaluate the contracts against.
3. Bindings that exist between the contracts and the IUT.

We will now examine the first of the four modular contracts. Following each contract listing a detailed description of the syntax and semantics will be presented. Each contract is expressed using Another Contract Language (ACL). The ACL is a simple high-level contract language. Details of which will be presented in a separate document. The ACL used in this example conforms to ACL 2.0, and includes features that were not present in the first ACL release.

The *Course* Contract

The Course contract represents a single university course. A course is created by the university, and consists of a name, code, a list of prerequisites, a list of students currently enrolled in the course, and a size limit on the number of students that can take the course. The contract listing is shown below:

```
Import Core;

Namespace DaveArnold.Examples.School
{
	Contract Course
	{
		Parameters
		{
			Scalar Boolean EnforcePreRequisites =
				{ true, default false };
			[0-2] Scalar Integer InstanceBind NumMidterms = 1;
			[0-5] Scalar Integer InstanceBind NumAssignments = 1;
			Scalar Boolean InstanceBind HasFinal =
				{ default true, false };
		}

		Observability String Name();
		Observability Integer Code();
		Observability List tStudent Students();
		Observability Integer CapSize();
		Observability List Integer PreRequisites();
		Observability Integer AssignmentWeight(Integer assignmentNum);
		Observability Integer MidtermWeight(Integer midtermNum);
		Observability Integer FinalWeight();
		Observability Integer MarkForStudent(tStudent student);

		Observability Boolean HasFinal()
		{
			Parameters.HasFinal == true;
		}

		Observability Integer TotalMarks()
		{
			Scalar Integer markTotal = 0;
			loop(1 to Parameters.NumAssignments)
			{
				markTotal = markTotal + AssignmentWeight(counter);
			}
			loop(1 to Parameters.NumMidterms)
			{
				markTotal = markTotal + MidtermWeight(counter);
			}
			choice(HasFinal()) true
			{
				markTotal = markTotal + FinalWeight();
			}
			value = markTotal;
		}
```

```
Observability Boolean IsFull()
{
	Students().Length() >= CapSize();
}

Observability Boolean HasPreRequisites(tStudent s)
{
	List Integer completedCourses = s.TakenCourses();
	value = true;
	each(PreRequisites())
	{
		value = value && completedCourses.Contains(iterator);
	}
}

Responsibility new()
{
	Post(Name() not= null);
	Post(Code() not= 0);
	Post(Students().Length() == 0);
	Post(TotalMarks() == 100);
}

Responsibility finalize()
{
	Pre(Students().Length() == 0);
}

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

stub Responsibility AddStudent(tStudent s)
{
	[Default] AddStudentNoPreReqCheck(s);
	[Parameters.EnforcePreRequisites == true]
		AddStudentPreReqCheck(s);
}

Responsibility AddStudentNoPreReqCheck(tStudent s)
{
	Pre(Students().Contains(s) == false);

	Post(Students().Contains(s) == true);
}

Responsibility AddStudentPreReqCheck(tStudent s)
{
	Pre(Students().Contains(s) == false);
	Pre(HasPreRequisites(s));

	Post(Students().Contains(s) == true);
}
```

```
            Responsibility RemoveStudent(tStudent s)
            {
                  Pre(Students().Contains(s) == true);

                  Post(Students().Contains(s) == false);
            }

            Scenario ReportMarks
            {
                  Trigger(observe(TermEnded)),
                  once Scalar Contract University u = instance;
                  each(Students())
                  {
                        u.ReportMark(context, iterator,
                              MarkForStudent(iterator));
                  },
                  Terminate(fire(MarksRecorded));
            }

            Exports
            {
                  Type tStudent conforms Student
                  {
                        University::tStudent;
                  }
            }
      }
}
```

The Course contract listing begins with an ***Import*** statement. ***Import*** statements are used to reference plug-in namespaces that will be used within the contract. It should be noted that the Validation Framework (VF) system maintains two separate namespace systems. One for plug-ins, and one for contracts. The rationale here is that plug-ins are external items that are not used in the same way as contracts, and thus are kept separate. The ***Import*** statement is used for referencing plug-ins, and the ***Using*** statement (not shown) is used for referencing contract namespaces. In the case of our Course contract we are referencing the plug-in "Core" namespace. The "Core" namesace will contain a set of frequently used plug-ins that will be shipped with the VF. The "Core" namespace would be analogous to the contents of the "stdio.h" file in the C language. Details regarding the contents of the "Core" namespace will be provided in seperae documentation. ***Import*** statements must be placed before any other type of statements.

Next, the ***Namespace*** keyword is used to denote a new namespace entry. The name of the namespace entry follows the namespace keyword. All our contracts for this example will be located in the "DaveArnold.Examples.School" namespace. A given namespace may contain any number of namespace entries, but all contracts defined within a given namespace must have unique names. That is, two contracts with the same name cannot exist within the same namespace. Each namespace entry may contain any number of contract or interaction declarations. However for our example, we will only have a single item within each namespace entry.

The contract proper begins with the use of the ***Contract*** keyword. The ***Contract*** keyword is followed by an identifier denoting the name of the contract. As previously stated all contract names within a single namespace must be unique. The contract name is used to reference the current contract in other locations, such as within relations and other contracts. The contract name is also used as the binding point to reference the contract when a binding is made between the contract and a type defined within the IUT. Contracts defined using the ***Contract*** keyword are not automatically bound to an IUT counterpart. Rather, they are bound only if they are referenced within the contract project. To enable automatic binding between the contract and the IUT, the ***MainContract*** keyword should be used. An example of the ***MainContract*** keyword will be presented later in this example. Once a contract is defined, the body of the contract is specified between opening and closing brace brackets ('{' and '}'). Within the body of the contract a number of different contract sections can be specified that define the actual contract. There is no ordering to the placement of the contract sections. We will now examine the sections that compose our Course contract.

The first section found within the Course contract is a parameters section. The parameters section is denoted by the use of the ***Parameters*** keyword. The ***Parameters*** keyword is then followed by a set of matching brace brackets denoting the body of the parameters section. Parameters are used to configure the contract. They are analogous to testing parameters used by other testing tools. Parameters can be used to specify different scenario paths, or simply to denote a constant value, such as the maximum number of students within a class. There is no limit to the number of parameter sections that can occur within a contract; the only restriction is that all parameters defined within a contract must have a unique name. Parameters defined in base contracts are automatically inherited in all subcontracts. Let's now look at the five parameter definitions provided by the Course contract in detail.

The first parameter begins with the ***Scalar*** keyword. The ***Scalar*** keyword indicates that the parameter stores a scalar value, as opposed to a list value (denoted by the ***List*** keyword). Next, the type of parameter is specified. In the case of the EnforcePreRequisites parameter, the parameter is of type Boolean. After the parameter type, the name of the parameter follows. The parameter name is used to reference the parameter within and outside of the contract. Next, an equals (=) sign is used to assign a value to the parameter. In the case of the EnforcePreRequisites parameter, we have provided a set of possible values: ***true*** and ***false***. The ***default*** keyword that precedes the ***false*** value denotes that ***false*** is the default value. If the ***default*** keyword is not specified on any values then no default value will be provided. In the case of the EnforcePreRequisites parameter, at binding time, a value will be requested from the user. The value will only requested once and will apply to all instances of the Course contract. That is, the parameter is static across all instances of the contract. The EnforcePreRequisites parameter will be used to determine if students registering in the course will have their prerequisites checked when they register. We will now look at a parameter that can have a unique value for each instance of the contract: NumMidterms.

The NumMidterms parameter will be used to denote the number of midterm exams that the current course has. The parameter declaration begins with range information. Range information is optional and is located before the parameter declaration. If no range information is specified then the

parameter can be assigned any value, constrained to the parameter type. The range is specified between matching square brackets ('[' and ']'). The range shown for the NumMidterms parameter is an integer range that indicates, acceptable values are between 0 and 2. That is, a course may have 0, 1, or 2 midterms. In the case of string or character parameter types, a regular expression can be placed between the square brackets to indicate expected parameter values. Following the range, the ***Scalar*** and ***Integer*** keywords are used to indicate that the NumMidterms parameter is a scalar value and is of the Integer type. Next, the ***InstanceBind*** keyword is used to indicate that each instance of the Course contract has a unique parameter value. That is, each course has a different number of midterm exams. At run-time a binding box will be provided to the user each and every time that a new course is created. The user must specify a value that satisfies the range condition (if any). Finally, following the parameter name a default value is assigned. In the case of the NumMidterms parameter, the default value is 1. It should be noted that if the default value does not satisfy the parameter's range, a compile-time error will be generated.

Next, the NumAssignments parameter is declared in a similar fashion, and will be used to indicate the number of assignments that must be completed during the course. One more parameter declaration follows: HasFinal. HasFinal is a scalar Boolean value that is bound per contract instance indicating if the course has a final exam. The default value for the HasFinal parameter is ***true*** (denoted by the ***default*** keyword).

Following the parameters section is a set of observability methods. Observability methods are used to acquire information from the IUT type that is bound to the contract. There are two types of observability methods: bound observability methods, and derived observability methods. Bound observability methods do not contain a body and are bound directly to a procedure within the IUT that has the same signature as the observability. The same signature is defined as the same return type, and parameter types (in any order). It should be noted that the name of the observability method does not have to match the name of the procedure that it is bound to. Instrumentation will be added to the bound IUT procedures to ensure that all observability methods are side-effect free. Derived observability methods are not bound to a procedure within the IUT. Their value is calculated by using other observability methods and contract variables. The Course contract contains examples of both types of observability methods. We will start by presenting the bound observabilities.

The first bound observability is called ***Name()***. The ***Name()*** observability will be bound to an IUT procedure that returns the name of the course. Likewise the ***Code()*** observability will be bound to an IUT procedure that returns a integer value representing the course code. Next, the ***Students()*** observability is defined. The return type of the ***Students()*** observability contains the ***List*** keyword in addition to the return type (tStudent). By default all observability methods return a scalar value. That is, the ***Scalar*** keyword is implied. The ***List*** keyword indicates that the ***Students()*** responsibility returns a list of type ***tStudent***. ***tStudent*** is a bound type. That is, the ***tStudent*** type is a placeholder for the IUT type that actually represents a student in the university. Details regarding the definition of the ***tStudent*** type will be provided shortly. The ***Students()*** observability is used to return a list of the students currently enrolled in the course. The ***CapSize()*** observability is used to return a scalar integer value indicating the maximum number of students that can be enrolled in the course. The ***PreRequsites()***

observability method will be bound to an IUT procedure that returns a list of integers indicating the course codes of all prerequisites required to take the current course. It should be noted that this list could be empty.

Next three observability methods are defined: ***AssignmentWeight()***, ***MidtermWeight()***, and ***FinalWeight()***. These observability methods are used to get the mark weight for each of the course components. In the case of the ***AssignmentWeight()*** and ***MidtermWeight()*** observability methods, a single parameter is also required by the observability method to indicate the assignment or midterm number that the caller is interested in. When binding observability methods that accept parameters, the IUT method must at least accept the requested parameter. By at least, we mean that the IUT method could accept additional parameters, and the values of these parameters must be specified by the user at bind-time.

Finally, a scalar integer observability method is defined named ***MarkForStudent()***. The ***MarkForStudent()*** observability is bound to an IUT method that returns the final mark for the student specified by the given student.

Next the Course contract specifies four derived observability methods. Recall that a derived observability method contains a body and the body determines the value returned to the caller. The first derived observability method is named ***HasFinal()*** and returns a Boolean value indicating if the course has a final exam or not. The resultant value is determined by referencing the ***HasFinal*** parameter. Parameters are referenced by using the ***Parameters*** keyword followed by the parameter name. The ***Parameters*** keyword is required to prevent ambiguity between parameters and variables. The ***HasFinal()*** derived observability method does not have an explicit ***value*** statement (return) because the body of the method contains a single expression. It is the result of this expression that is returned to the caller. We will now look at a derived observability method that requires an explicit ***value*** statement.

The ***TotalMarks()*** derived observability method is used to determine the total number of marks that can be obtained from this course. The body of the ***TotalMarks()*** observability method begins with the declaration of a scalar variable of type integer that has an initial value of zero. ACL 2.0 supports the use of initializing expressions in variable declarations. In the case of the ***markTotal*** variable, the initializing expression sets the variable to a value of zero. Next, the body contains one of the new statements found within ACL 2.0: a loop. As the name suggests, the loop statement is used to repeat a block of ACL code one or more times. The loop statement begins with the ***loop*** keyword, followed by the looping condition enclosed within matching round brackets (‘(‘ and ‘)’). The first element of the looping condition is the starting value. The starting value is the initial value of the loop counter and must be an integer value. The starting value is followed by the ***to*** keyword that indicates a separation between the starting value and the ending value. The ending value is the final value that the loop will operate on. That is, the final value is included in the loop iteration. If the final value is less than the starting value, the body of the loop will not execute. At each iteration of the loop, the loop counter, denoted by the ***counter*** keyword is increased by one. Nested loops are not allowed. Returning to the ***TotalMarks()*** derived observability method, the body of the observability begins by iterating through each assignment and adding its mark weight to the ***markTotal*** variable. If there are no assignments, the

body of the loop will not execute and no call to the ***AssignmentWeight*** bound observability will be made. Next, the same process occurs for any midterms that may exist within the course. Once the marks for any assignments and midterms have been added to our running total, we move on to a choice statement. A choice statement is similar to an if statement found within other programming languages, but it has been tailored for use at the requirements level. The choice statement begins with the use of the ***choice*** keyword, followed by the value that we are comparing against enclosed in matching round brackets ('(' and ')'). In the case of our observability method, the value returned by the previously discussed ***HasFinal()*** observability is used as the comparison value. Following the ending bracket, the choice condition is specified, in this case ***true***. The choice statement indicates that if the comparison value is equal to the choice condition, then the ACL code specified in the block immediately following the choice statement will be executed. If this is not the case, then the ACL code in the block will not be executed. Returning to the ***TotalMarks()*** derived observability, if there is a final exam in the course, then we add the value of the final exam to our running ***markTotal*** variable. Finally, because the observability body is not defined using a single expression, the resultant value has to be assigned to the ***value*** keyword to indicate the value that should be returned to the caller. If no such assignment exists, then a compile time error will be issued.

Next, the ***IsFull()*** derived observability is specified. The body of the ***IsFull()*** observability consists of a single expression that will return true if the course is full and no additional students can be registered in the course. The expression uses the built-in ***Length()*** operation defined on all list variables to return the number of students in the course. For the complete set of operations available on list variables please see the ACL specification. The number of students is compared with the cap size for the class to determine if the course is full or not.

Finally, the ***HasPreRequisites()*** derived observability method returns true if the given student has the required prerequisites to take the course. The body starts by defining a variable named ***completedCourses*** that is a list of integers. The variable's initializer assigns our new variable the list of courses that the student has already taken. Next, the observability body sets the ***value*** keyword to ***true***. Such an assignment is required because if the course has no prerequisites then the each statement will not execute and there will be no resultant value. Next, an each statement is used to iterate through each of the elements stored within the integer list returned from the ***PreRequsites()*** bound observability. The each statement is new in ACL 2.0 and begins with the ***each*** keyword, followed by the collection to iterate enclosed in round brackets ('(' and ')'). The current element of the iteration can be accessed using the ***iterator*** keyword. In ***HasPreRequisites()*** each course prerequisite is iterated through and checked against the list of courses that the given student has already completed. The value returned to the caller will be ***true*** if the course has no prerequisites or if the student has taken each of the required prerequisites and ***false*** otherwise.

The Course contract next defines a series of responsibilities that must be implemented by the IUT type bound to the contract. There are three ways that an IUT is able implement a given responsibility:

1) The responsibility is bound 1-to-1 to a method within the IUT.
2) The responsibility is bound to a starting method and an ending method. In this case the responsibility is said to be complete only after the ending method has completed execution.
3) The responsibility is decomposed into a grammar of other responsibilities.

Our example will illustrate all three responsibility types. The Course begins with one of two special responsibilities: ***new()***. As the name suggests the ***new()*** responsibility is automatically invoked upon the creation of a new instance of the IUT type bound to the contract. The ***new()*** responsibility does not require binding. The ***new()*** responsibility in the Course contract contains four post-conditions to ensure that a valid Course has been created. It should be noted that the ***new()*** responsibility cannot contain preconditions, as the IUT object has not been created before execution of a constructor. The post-conditions ensure that the course has a name, a valid course code, that it does not already contain any students, and the total marks assigned during the course add up to 100.

Analogous to the special ***new()*** responsibility is the special ***finalize()*** responsibility. As the name suggests the ***finalize()*** responsibility is automatically invoked right before an object of the IUT type bound to the current contract is destroyed. For that reason the ***finalize()*** responsibility cannot contain any post-conditions. In the Course contract our ***finalize()*** responsibility contains a single precondition that ensures that a course is not destroyed if it contains any students.

Next the Course contract defines an invariant named ***IsFullCheck***. An invariant is a way to ensure a consistent state at all times during the lifetime of the contract. Invariants are checked before and after each responsibility with two exceptions: Invariants are not checked upon entry to the ***new()*** responsibility or upon exit from the ***finalize()*** responsibility. In the case of the Course contract our invariant contains a single expression that will ensure that the course never ends up being over capacity. That is, having more students than the course size.

Following the ***IsFullCheck*** invariant, the Course contract defines a set of regular responsibilities that each and every course defined within the university is expected for perform. The first responsibility is ***AddStudent()***. As the name suggests the ***AddStudent()*** responsibility takes the given student and adds him/her to the course. However one look at the body of the ***AddStudent()*** responsibility doesn't seem to do that at all. Rather, the ***AddStudent()*** responsibility is a stub. Stub responsibilities are denoted by the use of the ***stub*** modifier before the responsibility declaration. Stub responsibilities are analogous to stubs found in use case maps: stubs represent a point where different functionality can be either statically or dynamically placed into the location represented by the stub. Put another way, a stub is like a place holder for ACL code. Stub responsibilities are new in ACL 2.0, and must contain a body. That is, stub responsibilities cannot be directly bound to an IUT procedure, but rather must be bound to another responsibility found within the contract. In the case of the ***AddStudent()*** responsibility we are defining a stub, because some courses may require a prerequisite check, where others may not, and may allow students without the required prerequisites to take a given course. This example represents a dynamic stub, in that the selection of which add student algorithm (prerequisite checking or not) will be done a runtime on a per course basis. Let us look at the body of the ***AddStudent()*** responsibility to see how this is accomplished with ACL 2.0. The body references two other responsibilities:

AddStudentNoPreReqCheck() and ***AddStudentPreReqCheck()*** . The selection of which one to execute is done using stub constraints. Stub constraints are specified between square brackets ('[' and ']') that indicates an expression that when it evaluates to true, the responsibility that follows the stub condition should be evaluated instead of the one defined by the stub. In addition, the ***Default*** keyword can be used to indicate a default choice of responsibility. In the ***AddStudent()*** responsibility, we do not perform a prerequisite check by default. As previously discussed, we have created a parameter whose value is specified on a per instance basis that indicates if the current course should check prerequisites or not. If the parameter 's value is set to ***true*** then, the ***AddStudentPreReqCheck()*** responsibility will be used. There are two things to note. First, if there is a case where more than one responsibility could be executed, based on their stub constraints, a runtime error will be issued. Secondly, a default responsibility is not required. If there is no responsibility that matches the stub constraints, then execution of the IUT proceeds as normal, and the ***AddStudent()*** responsibility is not executed. The fact that the responsibility is not executed is important when designing scenario grammars. Additional examples of stub responsibilities will be presented later in this document.

The ***AddStudentNoPreReqCheck()*** responsibility will be bound to an IUT procedure (or group of IUT procedures) that add a student to the course without checking prerequisites. The responsibility has one precondition that checks to ensure that student is not already in the class, and a post-condition to make sure that the student was actually added to the class.

The ***AddStudentPreReqCheck()*** responsibility is similar to the ***AddStudentNoPreReqCheck()*** with the exception that ***AddStudentPreReqCheck()*** contains an additional precondition that uses the previously defined ***HasPreRequisites()*** observability method to ensure that the student being registered in the course does in-fact have the required pre requisites.

Analogous to the ***AddStudent()*** responsibility, the ***RemoveStudent()*** responsibility removes a student from the course. The responsibility defines a single precondition that ensures that the student to be removed from the course has been previously enrolled in the course. Finally, the responsibility defines a post-condition to ensure that the after the bound IUT procedure has completed executing, the student is no longer enrolled in the course.

The Course contract next defines a single scenario: ***ReportMarks***. The scenario specifies a grammar for the process of recording a mark for each student enrolled in the course. A number of changes have been made to the scenario contract element in ACL 2.0. The ***ReportMarks*** scenario will outline a few of them. For a complete list see the ACL 2.0 specification document. The scenario begins with the ***Scenario*** keyword followed by the name of the scenario. The scenario name is used in the contract evaluation report and within the ***Relation*** construct. The ***Relation*** construct will be discussed later in this document. Scenarios do not accept parameters. The rationale here is that scenarios are triggered by observed events or IUT procedures being invoked, there is no way to invoke a scenario, and as such it cannot accept parameters. The body of the scenario is specified between matching brace brackets ('{' and '}'). The scenario begins with a trigger. As in ACL 1.0, the scenario trigger is specified using the ***Trigger*** keyword. The scenario trigger can be one of two things: responsibility or an observed event. A responsibility trigger occurs after the specified responsibility has completed executing. That is,

the IUT procedure (or group of) bound to the responsibility have finished executing, and any post-conditions have been evaluated. This was the only way of triggering a scenario in ACL 1.0. In ACL 2.0, there is an additional was to trigger a scenario, through the use of an observable event. An observable event trigger is denoted by using the ***observe*** keyword within the body of the ***Trigger*** statement, as shown in the ***ReportMarks*** scenario. The ***observe*** keyword is followed by the event name enclosed in matching round brackets ('(' and ')'). Event names are represented by identifiers, and follow the same naming rules as variables. Details regarding the observable event system within ACL 2.0 will be provided later in this document. Returning to the ***ReportMarks*** scenario, the scenario is triggered upon receipt of the event named ***TermEnded***. That is, the scenario begins once a term has ended.

Next the scenario captures the current University contract instance. In ACL 2.0, contracts are variable types. That is, instances of contracts can be referenced. Through these instances, contract variables, observability methods, and responsibilities can be used. The declaration of a contract variable begins with the ***Contract*** keyword followed by the name of the contract that we are referencing. Note that if the requested contract does not reside within the current contract namespace, a Using declaration is required before the current contract declaration. Following, the contract name, an identifier is used to represent the contract instance within the current scope, in our case within the body of the ***ReportMarks*** scenario. Contracts are instantiated only when their bound IUT type is instantiated. As such, new contract instances cannot be created within the ACL. There are two ways to obtain a contract instance.

The first way is illustrated in the ***ReportMarks*** scenario and uses the ***instance*** keyword to assign the contract variable the current instance of the contract. The ***instance*** keyword has different semantics depending on if the contract variable to be assigned is a scalar or list. If a scalar is used, the current instance is obtained using the following algorithm:

1. If there is no current contract instance of the requested type, the scenario fails.
2. If there is a single contract instance of the requested type, that instance is used.
3. If there is more than one contract instance of the requested type, the triggering event is examined to see if a specific contract instance was used to fire the triggering event. If so, then it is that instance that is used. In the case of the ***ReportMarks*** scenario if the algorithm gets to Step 3, then the university instance that fired the ***TermEnded*** event would be used.
4. Finally, if the runtime cannot determine the correct instance to use, the binding tool is used to request the instance to use from the user.

In the case, where the above algorithm is not sufficient in selecting the current contract instance, the ***instance*** keyword can be replaced with the ***bind*** keyword (not shown) to force the presentation of the binding tool for instance selection.

If a list is used, the current instance list is obtained using the following algorithm:

1. If there is no current contract instance of the requested type, an empty list of contract instances is used.
2. If there is a single contract instance of the requested type, a list with a single instance is used.
3. If there is more than one contract instance of the requested type, a list with each instance is used.

If the ***bind*** keyword is used for a list of contract instances, the binding tool will allow for multiple instance selection. The selected instances will then be placed into a list and assigned to the specified contract variable.

The second way to obtain a contract instance is through the use of the ***bindpoint*** keyword with an IUT type. The ***bindpoint*** keyword is used to obtain the element bound to a given item. The ***bindpoint*** keyword can only be used to obtain scalar contract instances and works as follows:

1. In <IUT type instance>.***bindpoint*** – The contract instance that the IUT instance has been bound to is returned. If there is no contract bound to the IUT type, a ***null*** value is returned.
2. In <contract instance>.***bindpoint*** – The IUT instance that has been bound to the contract instance is returned. This value cannot be null, because a contract cannot be created without having an underlying IUT type.

Examples of both uses of the ***bindpoint*** keyword will be presented in this document. To recap, the ***ReportMarks*** scenario is triggered following the receipt of the ***TermEnded*** event, and beings with acquiring the current instance of the University contract, using the algorithm above. Next, the scenario uses the previously discussed ***each*** statement to iterate through each student within the course. When the ***each*** statement is used within a scenario, it denotes that the scenario grammar specified within the body of the ***each*** statement must be executed for each element within the list. It should be noted that there is no ordering requirements, the students can be processed in any order as long as each student is processed, once and only once. In addition, it is possible that several collection elements can be processed in parallel. That is, the body of the ***each*** statement does not denote any concurrency constraints.

The body of the ***each*** statement contains a single grammar element. It denotes that the ***ReportMark*** responsibility defined on the university contract instance is invoked using our course, the current student, denoted by the ***iterator*** keyword, and that the mark reported by the previously discussed ***MarkForStudent()*** observability method is provided. The purpose of this element is to ensure that the correct mark for the given student is recorded. A positive scenario grammar element match is made only when the following conditions are true:

1. The IUT procedure(s) bound to the ***ReportMark*** responsibility have been executed on the IUT instance that is bound to the contract instance that is captured in the ***u*** contract variable. (The responsibility could be implemented by a grammar of other responsibilities, in this case the grammar is flattened, and then the step proceeds as normal).

2. Any pre/post-conditions and/or other constraints specified within the ***ReportMark*** responsibility have evaluated to true.
3. The parameters passed to the ***ReportMark*** responsibility are as follows:
    a. The first parameter is the instance of the IUT type bound to this course. Note: The ***context*** keyword should actually be ***context.bindpoint*** but, the compiler is able to infer when the IUT type should be used rather than the contract instance, so the ***bindpoint*** keyword is not required in the is case.
    b. The second parameter is the student specified by the ***iterator*** keyword. This is actually any student within the ***Students()*** list that has not already been used.
    c. The third parameter is the same value as returned by the ***MarkForStudent()*** observability.

Following the ***each*** statement, the scenario is completed with a termination condition. A scenario termination condition is used to specify when the scenario is complete. As with scenario triggers termination conditions have also been updated in ACL 2.0 to include support for observable events. A scenario termination condition begins with the ***Terminate*** keyword followed by one of four elements enclosed in round brackets ('(' and ')'):

1. A responsibility - This was the only way to specify a scenario termination condition in ACL 1.0. The scenario terminates when the specified responsibility has completed execution. Execution is defined as previously discussed with triggering events.
2. An observe statement – The scenario is completed upon receipt of the specified observable event.
3. A fire statement – The fire statement (shown here) is denoted by the ***fire*** keyword and is used to raise the event specified within matching round brackets ('(' and ')'). The fire statement is used to raise events that are observed using an observe statement.
4. Empty – An empty terminate statement is denoted with an empty set of matching round brackets ('(' and ')'), and indicates that the scenario automatically terminates when execution reaches the terminate statement.

The terminate statement completes the ***ReportMarks*** scenario. The Course contract continues with an Exports section. The Exports section defines the IUT binding points required for the contract, and also includes any binding constraints. Our Course contract only requires a single binding point: ***tStudent***. The binding point begins with the ***Type*** keyword to denote that the binding of the ***tStudent*** symbol is to be made against a type within the IUT. The ***Method***, and ***Field*** keywords can be used to bind a contract symbol to an IUT procedure or field respectively. Only the binding type keyword and the symbol are required to define a binding point. The ***tStudent*** symbol is followed by the ***conforms*** keyword. The ***conforms*** keyword is used to indicate that the IUT type that the ***tStudent*** symbol is bound to will automatically have the ***Student*** contract applied to it. Following the binding point, a set of binding rules/constraints can be optionally specified within an enclosing set of brace brackets ('{' and '}'). The ***tStudent*** binding point contains a single binding rule that states that the IUT type that is bound to the ***tStudent*** symbol must be the same type that is bound to the ***tStudent*** symbol found within the University contract. The Exports section concludes the Course contract. We will now look at a contract that offers a refinement on our Course contract by creating a Course that also includes a course project in addition to just the assignments and exams provided by the just presented Course contract.

The *ProjectCourse* Contract

The ProjectCourse contract represents a refinement on the Course contract representing a single university course. The ProjectCourse is used for courses that also include a course project in addition to any number of assignments, mid-terms, and a final exam. The contract listing is shown below:

```
Import Core;

Namespace DaveArnold.Examples.School
{
	Contract ProjectCourse extends Course
	{
		Parameters
		{
			Scalar Boolean InstanceBind HasProject =
				{ default true, false };
		}

		Observability Integer ProjectWeight();
		Observability Boolean HasProject()
		{
			Parameters.HasProject == true;
		}

		refine Observability Integer TotalMarks()
		{
			Scalar Integer markTotal = base.TotalMarks();
			choice(HasProject()) true
			{
				markTotal = markTotal + ProjectWeight();
			}
			value = markTotal;
		}
	}
}
```

The ProjectCourse contract listing begins with an ***Import*** statement to import the plug-ins that are located within the ”Core” namespace. Recall that the “Core” namespace will contain a set of commonly used plug-ins. Next a namespace declaration is used to place the ProjectCourse contract into the “DaveArnold.Examples.School” namespace. This namespace will be used for all of the contracts in our example. The ProjectCourse contract begins with the ***Contract*** keyword followed by the name of the contract. After the ProjectCourse contract identifier the ***extends*** keyword is used to denote a base contract. The name of the base contract follows the ***extends*** keyword. The contents of the base contract are available for use within the current contract. In the case of the ProjectCourse contract, we are using the Course contract to provide the functionality for a regular course. The ProjectCourse contract will then specialize to create a course that also contains a course project.

The ProjectCourse contract begins with a Parameters section. The parameters defined within the base contract are automatically part of the current project. ProjectCourse defines a new parameter named ***HasProject***. ***HasProject*** is a scalar Boolean value that defaults to ***true***. ***HasProject*** is used to define if the current course has a project. That is, it is possible for a project course to not actually have a project. For example a course may usually be a course project but during a given term the project portion of the course is cancelled.

Next, three observability methods are defined. The first observability method is named ***ProjectWeight()***. ***ProjectWeight()*** will be bound to a procedure within the IUT that will return an integer value representing the number of marks that compose the project component of the course. The second observability method is ***HasProject()*** . ***HasProject()*** is a defined observability method and will not be bound to a procedure within the IUT. The observability method's body consists of a single expression that tests the value of the previously discussed ***HasProject*** parameter to see if the current course has a project or not. The third and final observability method is a refinement over an existing observability defined within the base contract. It should be noted that only derived observability methods can be refined. Bound observability methods cannot be refined, as they do not contain a body. A refined observability method begins with the ***refine*** modifier before the ***Observability*** keyword. The ***refine*** modifier instructs the compiler that this observability method is refining an existing observability method. If no such base observability method exists, a compile-time error will be generated. The base observability method must have the same return type, name and parameter set as the refined observability. The CourseProject contract refines the ***TotalMarks()*** defined observability method so that the total mark calculation also includes the project mark. The body of the ***TotalMarks()*** observability method begins with the declaration of a scalar integer variable named ***markTotal***. ***markTotal*** will be used to record the total marks for the course. The variable's initializer uses the ***base*** keyword to access the base contract, and invoke the base version of the ***TotalMarks()*** observability. Such invocation gets the total marks for all aspects of the course except for the project. A ***choice*** statement is then used to execute the body of the ***choice*** statement only if the ***HasProject()*** observability method returns ***true***. The body of the ***choice*** statement consists of a single assignment that adds the weight of the course project, obtained from the ***ProjectWeight()*** bound observability method, to the ***markTotal*** variable. Finally, the ***TotalMarks()*** observability method assigns the ***markTotal*** variable to the ***value*** keyword so that the calculated total number of marks for the course is returned to the caller. The ***TotalMarks()*** observability method completes the ProjectCourse contract. It should be noted, that the ProjectCourse contract was included to provide an example of contract inheritance and refinement. Additional refinements could also be added to account for the complexities introduced within a project course. Or instead of creating a specialized ProjectCourse contract, the original Course contract could have been modified. Our example now turns away from courses, and moves to a contract representing an individual student.

The *Student* Contract

The Student contract represents an individual student enrolled at a university who is able to take courses. The Student contract listing is as follows:

```
Import Core;

Namespace DaveArnold.Examples.School
{
	Contract Student
	{
		Scalar Integer failures;

		Observability List Integer CompletedCourses();
		Observability Integer StudentNumber();
		Observability String Name();
		Observability List tCourse CurrentCourses();
		Observability Boolean IsFullTime();

		Observability Boolean IsCreated()
		{
			StudentNumber > 0;
		}

		Responsibility tCourse SelectCourse(List tCourse courses)
		{
			Post(value not= null);
			Post(courses.Contains(value) == true);
		}

		Responsibility RegisterCourse(tCourse course)
		{
			Pre(CurrentCourses().Contains(course) == false);

			Post(CurrentCourses().Contains(course) == true);
		}

		Responsibility DropCourse(tCourse course)
		{
			Pre(CurrentCourses().Contains(course) == true);

			Post(CurrentCourses().Contains(course) == false);
		}

		Responsibility DeRegisterCourse(tCourse course)
		{
			Pre(CurrentCourses().Contains(course) == true);

			Post(CurrentCourses().Contains(course) == false);
		}

		stub Responsibility DoAssignment(tCourse c)
		{
			Contract Course course = c.bindpoint;
			[course.Parameters.NumAssignments > 0] DoAssignment(c);
		}
```

```
stub Responsibility DoMidterm(tCourse c)
{
      Contract Course course = c.bindpoint;
      [course.Parameters.NumMidterms > 0] DoMidterm(c);
}

stub Responsibility DoFinal(tCourse c)
{
      Contract Course course = c.bindpoint;
      [course.Parameters.HasFinal == true] DoFinal(c);
}

stub Responsibility DoProject(tCourse c)
{
      Contract Course course = c.bindpoint;
      [course sameas ProjectCourse] DoProject(c);
}

Responsibility DoAssignment(tCourse c);
Responsibility DoMidterm(tCourse c);
Responsibility DoFinal(tCourse c);

Responsibility DoProject(tCourse c)
{
      FormATeam(c),
      observe(TeamFinalized),
      WorkOnProject(c);
}

Responsibility FormATeam(tCourse c)
{
      fire(TeamFinalized);
}

Responsibility WorkOnProject(tCourse c);
```

```
Scenario RegisterForCourses
{
	Scalar tCourse course;
	Contract University u = instance;
	Trigger(observe(CoursesCreated), IsCreated()),
	choice(IsFullTime()) true
	{
		(
			atomic
			{
				course = SelectCourse(u.Courses()),
				choice(course.bindpoint.IsFull()) true
				{
					course = SelectCourse(u.Courses()),
					redo;
				};
			},
			u.RegisterStudentForCourse(context, course),
			RegisterCourse(course)
		)[0-u.Parameters.MaxCoursesForFTStudents];
	}
	alternative(false)
	{
		(
			atomic
			{
				course = SelectCourse(u.Courses()),
				choice(course.bindpoint.IsFull()) true
				{
					course = SelectCourse(u.Courses()),
					redo;
				};
			},
			u.RegisterStudentForCourse(context, course),
			RegisterCourse(course)
		)[0-u.Parameters.MaxCoursesForPTStudents];
	},
	Terminate();
}
```

```
            Scenario TakeCourses
            {
                  failures = 0;
                  Trigger(observe(TermStarted)),
                  parallel
                  {
                        Contract Course course = instance;
                        Check(CurrentCourses().Contains(course.bindpoint));
                        atomic
                        {
                              (
                                    parallel
                                    {
                                          (DoAssignment(course.bindpoint))
                                          [course.Parameters.NumAssignments];
                                    }
                                    |
                                          (DoMidterm(course.bindpoint))
                                          [course.Parameters.NumMidterms]
                                    |

                                          (DoProject(course.bindpoint))
                                          [course sameas ProjectCourse &&
                                          course.Parameters.HasProject]
                              ),
                              (DoFinal(course.bindpoint)
                              [course.Parameters.HasFinal];
                        }
                        alternative(not observe(LastDayToDrop))
                        {
                              DropCourse(course.bindpoint);
                        };
                  }[CurrentCourses().Length()],
                  Terminate();
            }

            Exports
            {
                  Type tCourse conforms Course
                  {
                        University::tCourse;
                  }
            }
      }
}
```

The Student contract begins with the usual ***Import*** satement to import the "Core" plug-in namespace. Next, the ***Namespace*** keyword is used to denote a new namespace entry in the "DaveArnold.Examples.School" namespace. As previously stated, all of the contracts that compose the university example reside in the "DaveArnold.Examples.School" namespace. Within the namespace entry the Student contract begins. The contract begins with the ***Contract*** keyword followed by an identifer to denote the name of the contract, Student in our case. The body of the Student contract begins with the declaration of a variable named ***failures***. The ***failures*** variable is a scalar integer, and will be used to record the number of courses that the student has failed in the current term. The

number of failures is used to determine if a given student is able to continue with his/her degree. Usage of the ***failures*** variable will be shown shortly.

Next the Student contract defines five bound observability methods. The ***CompletedCourses()*** observability method will be bound to an IUT procedure that returns a list of integers representing all of the courses that have been completed by the student. Each integer represents the course code of a course that the student has completed in a previous term. The ***StudentNumber()*** observability method returns an integer that contains the student number for the current student. Likewise the ***Name()*** observability method returns a string representing the name of the current student. The ***CurrentCourses()*** observability method will be bound to an IUT procedure that returns a list of courses that the student is currently enrolled in. The ***tCourse*** type will be bound to the IUT type that represents a univeristy course. The binding point is defined within the Student contract's ***Exports*** section. Finally, the ***IsFullTime()*** bound observability method returns ***true*** if the current student is a full time student, and ***false*** otherwise (part time).

The Student contract then defines a single defined observability method, named ***IsCreated()***. As the name suggests, ***IsCreated()*** returns ***true*** if the student has been assigned a valid student number, ***false*** otherwise. The body of the ***IsCreated()*** observability method uses a single expression to test that the student has been assigned a student number. It is the result of this single expression that is returned to the caller.

The ***SelectCourse()*** responsibility is a black box responsibility that is bound to a procedure (or procedures) within the IUT that selects a course from the provided list of courses that the student wants to take. The selection algorithm is not of importance to our contract, but it could be specified using various ACL constructs. The ***SelectCourse()*** responsibility contains two post-conditions that ensure that the result returned from the bound IUT elements is not null and that the result is from the list of courses provided. That is, a valid course was actually selected.

Next, the ***RegisterCourse()*** responsibility is defined to register a student in a course. The responsibility defines a single precondition that checks to make sure that the student hasn't already registered in the course. That is, the student's list of current courses doesn't already contain the course that they want to register in. Next, a post-condition is used to ensure that the upon completion of the responsibility, the student is registered in the requested course.

The Student contract then defines the ***DropCourse()*** responsibility. The ***DropCourse()*** responsibility is a black box that represents the action of a student dropping a course. The course to drop is specified by the responsibility's only parameter. The responsibility defines a single precondition to ensure that the student is currently enrolled in the course that he/she wishes to drop. A corresponding post-condition checks to see that the student is no longer enrolled in the course that he/she chose to drop. That is, the course has been successfully dropped. The ***DeRegisterCourse()*** responsibility follows, and contains the same pre and post-conditions as the ***DropCourse()*** responsibility. The difference between the two responsibilities is that ***DropCourse()*** is used to drop a course that has already started, where as ***DeRegisterCourse()*** is used to remove the student from a

course that has not yet begun. Depending on how the IUT is implemented, the two responsibilities could be bound to the same IUT procedure. Such binding is not apparent from the contract point of view, and would only be discovered during execution of the binding tool.

Next a stub responsibility named ***DoAssignment()*** is defined. The ***DoAssignment()*** responsibility is a stub for a responsibility that will perform the task of a student doing an assignment. The rationale for using a responsibility stub in this situation is that is if a student is not taking a course that has any assignments, then the student will not require a corresponding responsibility. The body of the ***DoAssignment()*** responsibility stub begins with the extraction of the Course contract instance from the provided IUT instance representing the given course that we are doing the assignment for. The ***bindpoint*** keyword is used to get the Course contract instance that is attached to the IUT instance. Recall that the ***bindpoint*** keyword is used to get the opposite end of a binding. That is, a contract instance from an IUT instance, or an IUT instance from a contract. Note that if the resultant contract type does not match the declared type a run-time error will be issued. An example of such an error would be if a contract other than the Course contract was bound to the IUT type represented by the ***tCourse*** symbol. In addition, if the IUT instance does not have a corresponding contract instance, a value of ***null*** will be returned. The second line of the responsibility stub indicates the responsibility that should be used to fill the stub and a constraint for its use. Let us elaborate, the constraint specifies that the ***DoAssignment()*** regular (non-stub) responsibility will be used if the given course has at least one assignment. There are two things to note here. The first is that the Student contract contains two ***DoAssignment()*** responsibilities. One is a stub and the other is a regular responsibility. While such naming is not required, it is allowed so that a stub can be used to select possible responsibilities (including one with the same name) that can be invoked depending on the evaluation of a constraint. The second thing to note is what happens if there is no responsibility that fits into the stub, due to the constraint evaluation. In this case no responsibility executes and the responsibility stub is automatically completed. That is, if the responsibility stub is part of a scenario grammar the grammar element will automatically be satisfied.

The ***DoMidterm()***, ***DoFinal()***, and ***DoProject()*** responsibility stubs perform the similar action to the ***DoAssignment()*** responsibility. As such their specifics will not be discussed any further. One thing to note is that the constraint used in the body of the ***DoProject()*** responsibility uses the ***sameas*** keyword to check if the given course is actually an instance of the ProjectCourse contract. That is, the student can only do a project within a course that contains a project.

Following the four stub responsibilities, their corresponding regular responsibilities are defined. The ***DoAssignment()***, ***DoMidterm()***, and ***DoFinal()*** responsibilities will each be bound to an IUT procedure (or group of procedures), that carries out the assignment, mid-term or final task respectively. The responsibilities defined within the Student contract do not define any pre or post-conditions but they could be enriched with any of the constraint functionality that has already been discussed. Next the ***DoProject()*** responsibility is defined. The ***DoProject()*** responsibility is an example of a responsibility that is not bound to a corresponding IUT procedure, but rather is specified by a grammar of other responsibilities and events. The difference between the two types of responsibilities can be seen within the responsibility body. If the body, as in ***DoProject()***, contains a scenario grammar then a binding will

not be created between the responsibility and the IUT, but rather the responsibility will be marked as completed only following a successful execution of the containing grammar. It should be noted, that stub responsibilities cannot be defined using a scenario grammar. The rationale for such a restriction is that a stub responsibility is defined to be a place holder for one of possibly many responsibilities. The body of the stub responsibility will be replaced with the body of the selected regular responsibility. As such, a scenario grammar located within the body of the stub responsibility would be hidden by the responsibility used to fill in the stub. As such, stub responsibilities cannot be defined using a scenario grammar. If such a responsibility exists, a compile-time error will be generated. Returning to the ***DoProject()*** responsibility, the body defines a grammar of responsibilities for completing a course project. The course that the project is being done for is specified in the responsibility's only parameter. The grammar begins with the ***FormATeam()*** responsibility, followed by the observation of an event named ***TeamFinalized***. The ***TeamFinalized*** event will be used to indicate that the student has found a team to complete the project with. Following the observation of the ***TeamFinalized*** event, the ***WorkOnProject()*** responsibility is used to actually work on the project. The ***DoProject()*** responsibility is only completed if the specified scenario grammar is satisfied, this will only occur upon completion of the ***WorkOnProject()*** responsibility.

The Student contract then defines the ***FormATeam()*** responsibility. The dynamics of forming a team are treated as a black box in the context of our example, but they could be specified in detailed using a scenario grammar as was done in the ***DoProject()*** responsibility. In our case, the ***FormATeam()*** responsibility will be bound to an IUT procedure (or group of procedures) that performs the task of a student forming a team. The responsibility's body contains a single statement; the responsibility uses the ***fire*** keyword to send the ***TeamFinalized*** event upon completion of the responsibility. That is, when an event is fired from within a bound responsibility, the event is only fired upon completion of the responsibility. It should be noted that in the ***DoProject()*** responsibility, we did not need to explicitly observe the ***TeamFinalized*** event because when the ***FormATeam()*** responsibility completed the grammar would have moved on to the next element: ***WorkOnProject()***. However, we have included the event here to illustrate the use of the observable event system in ACL 2.0. More examples will be provided later in this document.

The final responsibility in the Student contract is ***WorkOnProject()***. The ***WorkOnProject()*** responsibility will be bound to an IUT procedure (or group of procedures) that will perform the task of actually working on a project for the course. We will examine the two scenarios that compose the Student contract.

The first scenario is named ***RegisterForCourses*** and as the name suggests is used to define the process of course registration as performed by the student. The scenario begins with the declaration of a scalar variable that will be used to store instances of courses. The course instances are of type ***tCourse***. The ***tCourse*** type will be bound to the IUT type that will represent a university course. Next, the current instance of the University contract is obtained using the previously discussed ***instance*** keyword. Following the variable declarations, the scenario beings with the scenario trigger denoted by the ***Trigger*** keyword. The trigger for the ***RegisterForCourses*** scenario is the observation of the ***CoursesCreated*** event. That is, a student begins course registration once the university has created all

the courses for the current term. Unlike the previously discussed scenario triggers, the trigger statement within the ***RegisterForCourses*** scenario contains an extra element: a constraint. Triggering constraints are specified by a Boolean expression that follows the actual triggering event as shown in the ***RegisterForCourses*** scenario. The triggering constraint must be ***true*** in order to trigger the scenario. If the triggering constraint does not result in a ***true*** value, the scenario is not triggered. It should be noted that unlike a precondition, an error is not generated, but rather the scenario simply is not triggered. If the triggering event occurs (again) at a later time and the triggering constraint now evaluates to ***true***, then the scenario will be triggered. Note, that such an occurrence requires the triggering event to re-occur. In the case of our example, the student would have a get a student number, but then would not be able to register until the next term, when the ***CoursesCreated*** event is fired. Returning to the ***RegisterForCourses*** scenario the triggering constraint is used to ensure that the student has in-fact been registered with the university and been assigned a student number, as per the previously discussed ***IsCreated()*** derived observability method. ***IsCreated()*** is a derived observability because rather than being bound to an IUT procedure, its result is calculated by the expression found within the body of the observability. Once the scenario has been triggered, a choice statement is used to denote a block of scenario grammar that is executed only if the current student is a full time student. The scenario block begins with the ***atomic*** keyword. Recall that an atomic block indicates a set of scenario elements, that are to be executed as one atomic action, and for the atomic block to be satisfied, the entirety of the body, specified between matching brace brackets ('{' and '}'), must be satisfied. For more information on the use of the ***atomic*** keyword see the ACL 2.0 specification document. The atomic block is required so that the action of selecting a course (regardless of the number of times the ***SelectCourse()*** responsibility is invoked) can be viewed as a single action from the scenario's point of view. The body of the atomic block begins with the invocation of the ***SelectCourse()*** responsibility with the list of courses provided by the University contract. The resulting (selected) course is then assigned to the previously declared ***course*** variable. The idea is that the value stored within in the ***course*** variable will be the course that the student wishes to register in. Following the course selection, the scenario continues with another choice statement. The choice statement checks to see if the selected course is full via the ***IsFull()*** responsibility defined on the Course contract. The ***bindpoint*** keyword is used to reference the contract instance that is bound to the IUT instance that has been assigned to the ***course*** variable. The body of the choice statement is then executed if the course that the student selected is full. That is, if the student cannot be registered in the course, because the course is full. The body begins with another invocation of the ***SelectCourse()*** responsibility to select another course. It should be noted, that our example has been designed with the assumption that the ***SelectCourse()*** responsibility will select a unique course each time that it is invoked. If this is was not the case, a new ***SelectDifferentCourse()*** responsibility could be created that accepts a list of courses that have already been selected, in addition to the list of university courses. Once a new course has been selected and assigned to the ***course*** variable, the body of the choice statement continues with the ***redo*** keyword. The ***redo*** keyword is new in ACL 2.0, and can only be used within the body of a choice, loop, or each statement. When the ***redo*** keyword is encountered, the runtime will return to the enclosing statement. In the case of a choice statement, the runtime will re-evaluate the choice condition. That is, the scenario grammar loops until the student selects a course that is not full. In the case of a loop or each statement, the ***redo*** keyword has the same functionality as the ***continue*** keyword found in C++ or

C#. The ***redo*** keyword only applies to the inner most choice, loop, or each statement. If no such statement exists a compile-time error will be issued. Returning to the ***RegisterForCourses*** scenario, once the student has selected a course that is not full, the scenario grammar specifies that the ***RegisterStudentForCourse()*** responsibility defined on the same contract instance that is bound to the ***u*** variable is to be invoked, using the supplied parameters. The first parameter is a reference to the current contract instance: the current student. Such a reference is accomplished using the ***context*** keyword. In fact, the correct syntax should be ***context.bindpoint***. However, as previously discussed the ***context*** keyword is automatically adapted to the correct side of the binding: the contract instance or the IUT instance. The second argument to the ***RegisterStudentForCourse()*** responsibility is the course to register the given student in. The course is specified by the ***course*** variable. After the student has been registered in the course with the university, the student's own ***RegisterCourse()*** responsibility is expected to complete the course registration process. As shown in the ***RegisterForCourses*** scenario, the grammar that selects and then registers the student in a course is enclosed within matching round brackets ('(' and ')'). Following the closing bracket is a range specification. A range specification is new in ACL 2.0. A range specification is placed within matching square brackets ('[' and ']'), and begins with an integer expression denoting the lower boundary of the range, followed by a dash ('-'), and finally another integer expression denoting the upper boundary. If there is no range, but rather a fixed value, the range specification can simply consist of a single integer expression. In the case of our scenario grammar the range specifies that the student may register is any number of courses (including zero) up to and including the maximum number of courses allowed for full time students. We know that the student is a full time student because the enclosing choice statement tested for full time status. The maximum number of courses allowed for a full time student is obtained from a parameter named ***MaxCoursesForFTStudents*** that is located within the University contract, denoted by the ***u*** variable. Now, what about part time students? Following the body of the choice statement that is an alternative. Each choice statement can have any number of alternatives, including zero. Each alternative is denoted by the ***alternative*** keyword, followed by a value to compare against the original choice expression enclosed in matching round brackets ('(' and ')'). Alternatives can are similar to case statements found in most object-oriented programming languages. There are a few things to note. First the original choice expression is only evaluated once. That is, the choice expression is not evaluated for each alternative. Next, an empty alternative can be specified as the last alternative. An empty alternative is denoted by the ***alternative*** keyword followed by the opening brace bracket for the body. That is, no value to compare against is specified, nor are the matching round brackets. Alternative statements can also be used in other locations, as will be illustrated in the next scenario. Returning to the ***RegisterForCourses*** scenario, the body of the alternative is evaluated if the ***IsFullTime()*** observability method returns ***false***. That is, the student is not a full time student, and thus part time. The body of the alternative is exactly the same as the body of the choice statement with one exception, the range specification uses the ***MaxCourcesForPTStudents*** parameter value used for the upper value, as opposed to the ***MaxCoursesForFTStudents***. Following the alternative, the ***RegisterForCourses*** scenario contains an empty terminate statement to denote the end of the scenario. Recall that a scenario termination event can be empty, denoted by an empty set of round brackets ('(' and ')') following the ***Terminate*** keyword. An empty terminate statement indicates that as soon as scenario execution reaches the terminate statement; the scenario grammar has been satisfied.

The Student contract continues with the ***TakeCourses*** scenario. The ***TakeCourses*** scenario denotes the grammar of responsibilities that must execute for the current student to actually take the courses that he/she enrolled in. The body of the scenario begins with setting the ***failures*** contract variable to zero. Recall that the ***failures*** variable is used to determine the number of courses that the student fails in the current term. As the ***TakeCourses*** scenario is used to denote the student taking the courses that he/she has enrolled in, the ***failures*** variable is re-set for the new term. Next a trigger statement is used to denote the triggering event for the scenario. The triggering event for the ***TakeCourses*** scenario is the observation of the ***TermStarted*** event. That is, the student begins taking courses when a new term starts. The University contract will fire the ***TermStarted*** event, and will be discussed shortly. Once the scenario is triggered, the ***parallel*** keyword is used to indicate a block of scenario grammar that can execute in parallel with itself. That is, several active sub-scenarios (denoted by the body of the parallel statement) can be executing at the same time. The purpose of the parallel block in the ***TakeCourses*** scenario, is that a student is able to take several courses simultaneously (bound by their student status, and university regulations regarding course loads). The body of the parallel block begins with the acquisition of a contract instance representing a Course contract. The idea here is to get a course that the student is currently taking. Next a check statement is used to ensure that the Course contract instance is bound to an IUT instance (via ***bindpoint***) that the student is currently enrolled in. That is, the current student is actually taking the course. Once the course is validated, an atomic block is used to denote that the action of completing the course requirements can be viewed as an atomic action with respect to the completion of a course. That is, the atomic statement ensures that a course can only be completed if each element of the course is completed. Within the body of the atomic statement there are three elements that have been 'or'ed ('|') together. Intuitively, it may seem that the three elements should be 'and'ed ('&') together, as the three elements all need to be completed before the final exam is written. However, the 'or' is used because the elements can be completed in any order. The range ('[n]') specified on each of the elements will ensure that each of the three elements is executed the correct number of times. The first element is enclosed within a parallel statement and represents a student doing assignments for the course. The assignments are within a parallel block because a student could be working on more than one assignment at a time. As denoted by the range, following the specification of the ***DoAssignment()*** stub responsibility, the student must complete the same number of assignments that the course has. The second element, consists of the ***DoMidterm()*** stub responsibility that must be completed the same number of times that the course has midterm exams. Notice that a parallel block is not used here, because it is not possible for a student to write two midterm exams as the same time. Finally, the ***DoProject()*** stub responsibility is used to complete the course project. Notice the use of a Boolean expression within the range specification. If the Boolean expression evaluates to ***true*** then, the range specification will result in a value of one, zero otherwise. That is, if the course is a project course and that the course has a project. The course type is tested using the previously discussed ***sameas*** keyword, and the ***HasProject*** parameter is used to see if the course project actually defines a course. Once each of the course elements have been completed by the student that is the assignment(s), midterm(s), and project (if any) is completed the student then writes the final exam. The final exam is expressed by the ***DoFinal()*** stub responsibility. If the ***HasFinal*** parameter contains a false value, then the range will have a value of zero, and the responsibility will be skipped. As previously stated the atomic block represents the task of completing the course. Following

the atomic block is an alternative. The alternative, denoted by the ***alternative*** keyword, indicates that at any point during execution of the preceding atomic block, the grammar specified by the alternative block may execute. Unlike an alternative for a choice statement, where only the value is required, an alternative that is not tied to a choice statement, must contain a complete expression, that when it evaluates to true, allows the body of the alternative to execute. In the ***TakeCourses*** scenario the alternative may execute anytime before the last day to drop a course. That is the ***LastDayToDrop*** event has not been observed. The body of the alternative contains a single responsibility, ***DropCourse()***. As the name suggests, the ***DropCourse()*** responsibility is used when a student wishes to drop a course. The alternative allows the student to drop a course at any time while the student is taking courses, up until the drop date. The parallel statement defining the grammar for taking or dropping a single course is repeated once for each course that the student is enrolled in. Once each of the courses that the student is enrolled in is either completed or dropped, the scenario automatically terminates due to the empty terminate statement, denoted by the ***terminate*** keyword followed by an empty set of matching round brackets (‘(‘ and ‘)’).

The final section in the Student contract is an exports section. The exports section denoted by the ***Exports*** keyword and a set of matching brace brackets (‘{‘ and ‘}’) defines a single binding point named ***tCourse*** that will be bound to an IUT type that will automatically have the previously discussed Course contract applied to it. It should be noted that any derivation of the Course contract, such as the ProjectCourse contract can be used. In the case where multiple derived Course contracts exists, the binding tool will request that the user select the contract that they want to bind to the IUT type that is bound to the ***tCourse*** symbol. In addition, the ***tCourse*** binding point includes a single binding constraint that states the ***tCourse*** symbol must be bound to the same IUT type that is/will be used to bind the ***tCourse*** type in the University contract. The exports section completes the Student contract. We will now examine the final contract of our example: the University.

The *University* Contract

The University contract represents the university itself. It will contain courses and students. The University contract also contains scenarios that will define university operation. The University contract is as follows:

```
Import Core;

Namespace DaveArnold.Examples.School
{
      MainContract University
      {
            Parameters
            {
                  [1-100] Scalar Integer InstanceBind UniversityCourses;
                  Scalar Integer MaxCoursesForFTStudents = 4;
                  Scalar Integer MaxCoursesForPTStudents = 2;
                  Scalar Integer PassRate = 70;
                  [1-12] Scalar Integer InstanceBind NumTermsToComplete;
            }

            Observability List tCourse Courses();
            Observability List tStudent Students();

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

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

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

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

            Responsibility RegisterStudentForCourse(tStudent student,
                                                    tCourse course);

            Responsibility CancelCourse(tCourse course)
            {
                  Pre(Courses().Contains(course) == true);

                  Post(Courses().Contains(course) == false);
            }
```

```
Responsibility DestroyCourse(tCourse course)
{
      once Scalar Integer oldSize;
      oldSize = PreSet(Courses().Length());
      Pre(Courses().Length() > 0);
      Pre(Courses().Contains(course) == true);

      Post(Courses().Length() == oldSize - 1);
      Post(Courses().Contains(course) == false);
}

Responsibility tStudent CreateStudent(String name);
Responsibility DestroyStudent(tStudent student);

Responsibility TermStarted();
Responsibility LastDayToDrop();
Responsibility TermEnded();

Responsibility CalculatePassFail()
{
      each(Students())
            choice(iterator.bindpoint.failures) >= 2
                  FailStudent(iterator);
            alternative
                  PassStudent(iterator);
}

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

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

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

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

```
            Scenario Term
            {
                  Trigger(new()),
                  (
                        CreateCourse()[Parameters.UniversityCourses],
                        TermStarted(),
                        fire(TermStarted),
                        LastDayToDrop(),
                        fire(LastDayToDrop),
                        TermEnded(),
                        fire(TermEnded),
                        observe(MarksRecorded)[Parameters.UniversityCourses],
                        CalculatePassFail(),
                        DestroyCourse()[Parameters.UniversityCourses],
                        fire(TermComplete)
                  )+,
                  Terminate(finalize());
            }

            Exports
            {
                  Type tCourse conforms Course
                  {
                        Student::tCourse;
                  }
                  Type tStudent conforms Student
                  {
                        Course::tStudent;
                  }
            }
      }
}
```

The Univesity contract represents a physical university and defines the creation of coruses and students. The University contract also specifiies several scenarios that govern how the physical university operates. The contract begins with the usual "Core" import statement followed by a namespace entry, denoted by the ***Namespace*** keyword, that places the University contract in the "DaveArnold.Examples.School" namespace. Unlike the previously discussed contracts, the University contract is specified using the ***MainContract*** keyword instead of the ***Contract*** keyword. Each contract project must contain at least one main contract. Main contracts behave and can have the exact same sections as a regular contract. The only difference is that a main contract must be bound to an IUT type at compile time, where regular contracts are only bound on demand. That is, if a regular contract is not referenced by any other bound countracts within a given contract project it will not be used, and the binding tool will not be invoked.

The body of the University contract begins with a parameter set, denoted by the ***Parameters*** keyword followed by matching brace brackets ('{' and '}'). The first parameter ***UniversityCourses*** is a scalar integer value that will denote the number of courses that the university will have each term. The range specified immediately before the parameter declaration specifies that the university must have between one and one-hundred courses. The actual value of the ***UniversityCourses*** parameter will be provided by the user through the binding tool, as per the use of the ***InstanceBind*** keyword. The next

three parameters have values that are hard-coded into the contract. That is, the user will not be asked by the binding tool to provided values for the parameters. The ***MaxCoursesForFTStudents*** parameter is a scalar value that indicates the maximum number of courses that a full time student can take per term. The ***MaxCoursesForPTStudents*** indicates the same value for part time students. The scalar integer ***PassRate*** stores the minimum mark required in order to pass a course. The final parameter, ***NumTermsToComplete***, is a scalar integer that is specified by the user through the binding tool that indicates the number of terms that each student must complete at the university in order to obtain a degree. As per the provided range this number can be between one and twelve terms.

Following the parameter definitions the University contract defines two bound observability methods. The first is named ***Courses()*** and returns a list of courses. The observability method will be bound to an IUT procedure that will return the list of courses that are currently created within the university. The second bound observability method is named ***Students()*** and returns a list of students. Like the ***Courses()*** bound observability, the ***Students()*** bound observability will be bound to a IUT procedure that returns a list of all the students that are currently enrolled at the university.

The University contract then defines a series of responsibilities. The first responsibility is the special ***new()*** responsibility that will be executed upon successful creation of a new instance of the IUT type bound to the University contract. The responsibility defines two post-conditions that ensure that the lists returned by the ***Courses()*** and ***Students()*** responsibilities do not contain any elements. That is, the university does not begin with a set of unknown courses or students. Likewise, the ***finalize()*** responsibility is executed immediately before the IUT instance bound to the contract is destroyed. The ***finalize()*** responsibility contains two preconditions that ensure that all students and courses have been removed from the university prior to destruction.

Next the ***CreateCourse()*** responsibility is defined. As the name suggests, the responsibility accepts a course name, and an integer representing the maximum number of students that can take the course, and returns a new course object of type ***tCourse***. The ***tCourse*** type is bound to an IUT type representing a course, as per the bind point definition in the contract's exports section. The body of the ***CreateCourse()*** responsibility begins with the definition of a scalar integer named ***oldSize***. The variable's definition includes the ***once*** modifier. The ***once*** modifier indicates that that the variable can only be assigned to once. If the variable is assigned to more than once a compile time error will be issued. The ***oldSize*** variable is assigned to using a preset statement. The preset statement is denoted by the ***PreSet*** keyword followed by an expression enclosed in a set of matching round brackets ('(' and ')'). Any preset statements are evaluated before the IUT procedure (or group of procedures) bound to the responsibility are invoked. That is, the preset statement is used to capture a value before execution of the IUT, so that the value can be used in a post-condition or other expression within the responsibility following IUT execution. In the case of the ***CreateCourse()*** responsibility, the ***oldSize*** variable is assigned the number of elements that exist within the course list returned from the ***Courses()*** observability method. Next, four post-conditions are specified to validate the creation of the course, and addition to the list of courses maintained by the university. The ***value*** keyword is used to reference the result returned by the bound IUT procedure. In the case where the responsibility is bound to a group of IUT procedures, the ***value*** keyword holds the value returned by the last IUT procedure in the set. The ***Name()*** and ***CapSize()***

observability methods defined in the Course contract are used to ensure that they return values that are consistent with the supplied parameters to the ***CreateCourse()*** responsibility. The last two post-conditions ensure that the course list has increased in size by one, and that the new course is contained within the list.

The ***RegisterStudentForCourse()*** responsibility follows, the responsibility will be bound to an IUT procedure (or group of procedures) that perform the task of registering the given student in the given course. The ***CancelCourse()*** responsibility will be bound to an IUT procedure (or group of procedures) that will cancel a course that has been previously created by the ***CreateCourse()*** responsibility. The course to cancel is provided as the single parameter to the responsibility. ***CreateCourse()*** specifies one precondition that ensures that the requested course to cancel is actually part of the current course list. The responsibility then uses a post-condition to ensure that the course is in-fact removed from the university. The University contract continues with the ***DestroyCourse()*** responsibility. The ***DestroyCourse()*** responsibility is used upon the completion of a term to remove a course from the list of courses, and to free up any resources that the course may have used. The body of the ***DestroyCourse()*** responsibility begins with the definition of a scalar integer named ***oldSize***. The ***once*** modifier indicates that the variable can only be assigned to once. Next, a preset statement is used to assign the number of courses stored by the university in the ***oldSize*** variable before execution of the bound IUT procedure(s). Two preconditions are then defined to ensure that the university course list contains at least one course, and that the given course to destroy is one of the courses in the list. ***DestroyCourse()*** then specifies two post-conditions that ensure that the given course was removed from the course list and that the course list contains one less element.

With responsibilities for the creating, registering in, canceling, and destroying of courses defined we will now look at responsibilities that apply to students. The first such responsibility is ***CreateStudent()*** . The ***CreateStudent()*** will be bound to an IUT procedure (or group of procedures) that creates a new student instance using the provided student name. The ***CreateStudent()*** responsibility does not define any pre or post-conditions. Such constraints could be specified to check the validity of the student name, and resultant student in as similar fashion as was done for the ***CreateCourse()*** responsibility. However, due to the fact that such constraints would not introduce any additional ACL functionality they have been omitted. Such an example, also illustrates how an ACL contract can initially be absent of detailed constraints and specifications, and then over time the contracts can be refined by the contract writer to yield a more detailed contract set. Analogous to the ***DestroyCourse()*** responsibility, the ***DestroyStudent()*** responsibility is used to remove and clean up a student from the university. Such an operation would occur once a student has graduated, or if the university was destroyed for any reason.

The University contract then defines a series of responsibilities for handling university operations during the duration of an academic term. The ***TermStarted()*** responsibility is invoked when classes for the current term begin. The invocation is defined by the execution of the IUT procedure (or group of procedures) that are bound to the ***TermStarted()*** responsibility. The ***LastDayToDrop()*** responsibility is bound to an IUT procedure (or group of procedures) that will be invoked when the last day to drop a course has occurred. After this day, students are unable to drop any courses that they are

enrolled in. The previously discussed Student contract enforces this rule. Next the ***TermEnded()*** responsibility is used to denote when the current university term has ended. Any end of term operations that would be performed by the university would be executed by the IUT procedure(s) bound to the ***TermEnded()*** responsibility.

Next, the University contract defines a derived responsibility named ***CaculatePassFail()***. Recall that instead of being bound to an IUT procedure (or a group of procedures) a derived responsibility is defined by the grammar specified in the body of the responsibility. In the case of the ***CalculatePassFail()*** responsibility the body consists of an each statement, that iterates through each student, denoted by the ***iterator*** keyword, to see if that student has passed or failed the term. A student is said to have failed the term if they have failed two or more courses. Recall from the Student contract that the number of failures is stored within the ***failures*** contract variable. A choice statement is used to compare the number of failures the student being iterated currently has. Because the list of students returned by the ***Students()*** observability method are of the ***tStudent*** type, the ***bindpoint*** keyword is used to obtain the contract instance to access the ***failures*** contract variable. Notice the use of the greater than or equals ('>=') operator in the choice statement. The previously discussed choice statements have not used an operator. That is because the equality ('==') operator is the default operator and does not have to be specified. However, in the case where you would like to use a different comparison operator, the operator must be specified. If the choice statement evaluates to ***true*** then the ***FailStudent()*** responsibility is expected to perform the actual action of failing the given student, denoted by the ***iterator*** keyword. An empty alternative statement is then specified to create an 'else' case. That is, if the choice statement evaluates to ***false*** then the body of the alternative statement will be executed. The body expects an invocation to the ***PassStudent()*** responsibility. The alternative completes the grammar of the ***CalculatePassFail()*** responsibility. Next, the actual ***FailStudent()*** and ***PassStudent()*** responsibilities are specified. Each responsibility will be bound to an IUT procedure (or group of procedures) that will perform the actual task of failing or passing a student respectively.

The final responsibility within the University contract is named ***ReportMark()*** and is used to report a mark for a student within a given course. The course, the student, and the mark to be recorded are provided as parameters to the responsibility. ***ReportMark()*** will be bound to a corresponding IUT procedure (or group of procedures) that will record the actual mark for the student. It should be noted that each parameter from the responsibility is mapped by the binding tool to a parameter in the IUT procedure. In the case where multiple IUT procedures are specified, the responsibility parameters can be bound as required by the tester to the parameters used by the procedures in the IUT set. That is, one responsibility parameter could be bound to the first IUT procedure, and a different parameter could be bound to the second IUT procedure. Or even, the same responsibility parameter could be bound to several IUT procedures. For more information on the binding process, please see the binding software development kit documentation.[1] The body of the ***ReportMark()*** responsibility contains a choice statement that checks the mark to be recorded against the previously discussed ***PassRate*** parameter. If the student has not received a sufficient mark to pass the course, the number of failures contract

[1] Not until the summer.

variable, ***failures***, is increased by one. With the ***ReportMark()*** responsibility completed, we will now turn our attention to the scenarios that compose the University contract.

The first scenario is named ***CreateCourses()***, and it is used to define the process that the university uses to create the courses at the start of a term. The body of the scenario begins with a trigger statement. The scenario trigger is the completion of the ***new()*** responsibility. Using the ***new()*** responsibility as a scenario trigger, has the effect of triggering the scenario as soon as a new IUT instance of the type bound to the contract has been created. The next step in the scenario grammar is the specification of the ***CreateCourse()*** responsibility, the range specified within matching square brackets ('[' and ']') specifies that the ***CreateCourse()*** responsibility must be invoked once for each course that university is offering this term, denoted by the ***UniversityCourses*** parameter. Once all of the university courses have been created, the scenario grammar moves on to the terminate statement. The terminate statement fires the ***CoursesCreated*** event. That is, the ***CoursesCreated*** event will be fired when all of the requested university courses have been created and added to the list of courses offered by the university.

Next, the ***CreateStudents*** scenario is used to specify how and when students can be created. The scenario is triggered using the ***new()*** responsibility, and thus is triggered as soon as the university is created. Next, one or more students are created through the ***CreateStudent()*** responsibility. The plus ('+') is used to indicate that one or more students can be created and added to the university. The scenario grammar is completed by the terminate statement. The ***CreateStudents*** scenario is terminated upon successful execution of the ***finalize()*** responsibility. That is, the scenario ends when the university is destroyed.

Finally, the University contract defines the ***Term*** scenario. The ***Term*** scenario is used to define a grammar representing how a term is managed by the university. The body of the scenario begins with a trigger statement, the ***Term*** scenario is triggered by the ***new()*** responsibility. Next, the grammar begins with the ***CreateCourse()*** responsibility. The grammar specifies that ***CreateCourse()*** is required to execute once for each course that the university wants to create, denoted by the ***UniversityCourses*** parameter. Once all the courses have been created, the grammar then specifies that the ***TermStarted()*** responsibility is then executed. Upon completion of the ***TermStarted()*** responsibility the grammar moves on to a fire statement. The fire statement, denoted by the ***fire*** keyword, will immediately fire the ***TermStarted*** observable event when the scenario grammar encounters the statement. That is, as soon as the ***TermStarted()*** responsibility finishes executing the ***TermStarted*** event is fired. Next, the responsibility waits for the ***LastDayToDrop()*** responsibility to finish executing, and then fires the ***LastDayToDrop*** observable event. The grammar then specifies the same sequence of events with the ***TermEnded()*** responsibility and the ***TermEnded*** event. After the ***TermEnded*** event is fired, the scenario grammar waits for the observation, denoted by the ***observe*** keyword, of the ***MarksRecorded*** observable event. The range specified between matching square brackets ('[' and ']') specifies that the scenario must observe the ***MarksRecorded*** event for each course in the university, denoted by the ***UniversityCourses*** parameter. Next, the grammar specifies the execution of the ***CalculatePassFail()*** responsibility, followed by the ***DestroyCourse()*** responsibility for each course that was created. Once all the courses have been destroyed using the ***DestroyCourse()*** responsibility, the scenario grammar fires

the ***TermComplete*** observable event. The entire operation of a university term is enclosed within matching round brackets ('(' and ')') followed by a plus ('+'). Such notation denotes that during execution of the IUT, several terms may occur. However, the university cannot be destroyed during the execution of a term. Finally, the scenario grammar specifies a terminate statement, the ***Term*** scenario only terminates upon successful execution of the ***finalize()*** responsibility. That is, the ***Term*** scenario only terminates when the university is destroyed.

The final section of the University contract is an exports section. The exports section is denoted by the ***Exports*** keyword followed by matching brace brackets ('{' and '}') that contain the body of the exports section. The exports body in the University contract defines two binding points. One that binds the ***tCourse*** symbol to an IUT type that conforms to the Course contract, the other binds the ***tStudent*** symbol to an IUT type that conforms to the Student contract. Both binding points contain constraints, so that the IUT type used for the course and the student is consistent throughout the contract project. The exports section completes the University contract.

The University contract concludes the set of contracts that compose our university example. While, contracts specify responsibilities and scenarios that operate within the context of the containing contract, there is no global specification illustrating the interactions between contracts. Such global specification is analogous to a use case diagram used to specify the interactions between use cases that compose a system's specification. In ACL 2.0, such a notation was added in the form of a new first order entity known as an interaction. Interactions are optional, and a given contract project can contain any number of interactions. Our university contains a single interaction that we will now discuss.

The *School* Interaction

```
Import Core;

Namespace DaveArnold.Examples.School
{
	Interaction School
	{
		Relation Creation
		{
			Contract University u;
			(u.CreateStudents || u.CreateCourses);
		}

		Relation Cancelling
		{
			Contract University u;
			Instance c;
			c = u.CreateCourse(dontcare, dontcare),
			(u.CancelCourse(c))?,
			observe(TermStarted);
		}

		Relation Students
		{
			Contract Student s;
			Contract University u;
			(
				s.RegisterForCourses,
				observe(TermStarted),
				s.TakeCourses,
				observe(CourseComplete)[s.CurrentCourses().Length()],
				observe(TermEnded),
				observe(MarkRecorded)[s.CurrentCourses().Length()],
				observe(TermComplete)
			)[u.Parameters.NumTermsToComplete];
		}
	}
}
```

Like contracts interactions must reside within a namespace entry. The School interaction is is located within the "DaveArnold.Examples.School" namespace, as were the other contracts defined in our example. The interaction begins with the ***Interaction*** keyword followed by an identifier representing the name of the interaction. Each contract project may contain zero or more interactions. Interaction names must be unique within a given namespace, and are used in the contract evaluation report to identify the interactions. However, other than identification purposes, the interaction name is not used. The body of an interaction is specified with a set of matching brace brackets ('{' and '}'). As with contracts, the body of an interaction contains a set of sections. Interactions can only contain a single type of section: a relation. Each interaction may contain any number of relations. As the name suggests, a relation is used to specify how scenarios and responsibilities defined within contracts relate

and interact with one another. A relation is denoted with the ***Relation*** keyword followed by an indentifier representing the name of the relation. Each relation defined within an interaction must have a unique name. Relation names are used for identification purposes only within the contract evaluation report and to not have any additional meaning. The body of a relation is defined by a set of matching brace brackets ('{' and '}'). The School relation defines three relations.

The first relation, ***Creation***, is used to define how scenarios used to create the university interact. The body of the ***Creation*** relation begins with the delcaration of a contract variable. Unlike variables defined within a contract, relation contract variables do not have an instance. That is, all instances of a given contract must obey any relation that is defined. A relation contract variable is defined by the ***Contract*** keyword followed by the name of the contract, and an identifier denoting the varaible name. The body of the ***Creation*** relation continues with a relation grammar. A relation grammar is similar to the scenario grammars found within a contract with the following exceptions:

1. Relations do not have triggering or termination events.
2. The ***atomic***, ***parallel***, or ***fire*** keywords and corresponding statements cannot be used in relations.
3. Scenarios and responsibilities can both be used as part of the relation grammar.

Returning to the ***Creation*** relation, the relation grammar specifes that the ***CreateStudents*** and ***CreateCourses*** scenarios defined in the University contract are independent and their execution can overlap. Such a relation is expressed usign the independence operator ('||'). For more details regarding the independence operator see the ACL 2.0 specification document. This simple independence relation completes the ***Creation*** relation.

Next the School interaction defines the ***Cancelling*** relation. The ***Cancelling*** relation is used to specify that the university can only cancel courses until the term has started. That is, once the term has started, the course can no longer be cancelled. The body of the relation begins by creating contract varaible, ***u***, represeting the University contract. Next, the ***Instance*** keyword is used to define an instance variable, ***c,*** that will be used to store the course that the university has created. Instance variables are used to represent values returned and sent (via parameters) to responsibilities. Instance varibles are denoted by the ***Instance*** keyword followed by an identifier that will be used to represent the variable. Instance variable declarations do not specify a type, because their type is inferred based on the instance variable's first use. In the case of the ***Cancelling*** relation, the instance variable will be of type ***tCourse***, as per the definition of the ***CreateCourse()*** responsibility. The ***dontcare*** keywords are used to indicate that the parameters passed to the ***CreateCourse()*** responsibility are not of interest to the ***Cancelling*** relation. Next, the ***CancelCourse()*** responsibility is specified followed by the optional operator ('?'). As the name suggests, the optional operator is used to denote that the relation grammar that is specified before the optional operator is optional and does not have to execute in order for the relation to be satisified. In addition, the optional operator only allows the corresponding relation grammar to be executed at most once, that is repetition of the relation grammar is not permitted. Finally, the ***Cancelling*** relation concludes with the observation of that ***TermStarted*** event. That is, the

relation specifies that for each course that is created within the university the course may be optionally cancelled before observation of the ***TermStarted*** event.

The ***Students*** relation completes the definition of the School interaction. The ***Students*** relation is used to enforce the order of execution for the scenarios defined within the Student contract. The body of the relation begins with the definition of two contract variables. One, ***s***, to represent the Student contract and one, ***u***, to represent the University contract. The relation grammar indicates using the follows operator (',') that the ***RegisterForCourses*** scenario must execute before the observation of the ***TermStarted*** event that must occur before the ***TakeCourses*** scenario, etc. The entire block of relation grammar must execute once for each term that the university requires the student to complete, denoted by the ***NumTermsToComplete*** parameter defined by the University contract. It should be noted that a separate relation instance will be created by the runtime for each student contract instance that is created. Thus, the ***Students*** relation ensures that each student within the university completes the required number of terms. The ***Students*** relation completes the School interaction. The School interaction completes the university example.

Step 2 – Contract Compilation

The first step in processing the contracts and interaction listed above is to create a contract project. As previously stated, contract projects are implemented as a project type within Visual Studio 2008, and consist of the following elements:

- One or more contracts with at least one main contract.
- Zero or more interactions.
- An IUT that represents the system being modeled by the contracts and interactions.
- A set of bindings to bind the contracts to the IUT

The entire contract project is sent to the ACL compiler that tokenizes, and parses the contracts and interactions. The compiler flattens any contract inheritance, performs generic parameter substitution, and ensures that all identifiers can be resolved. That is, the ACL compiler ensures that the contract syntax is correct, and that all required plug-ins have been located. At this point the contracts and interactions are represented by an abstract syntax tree. Each requested binding has been mapped to any other bindings based on the specified binding rules. The next step is to perform the actual bindings.

Step 3 – Bindings

As the name suggests, the types, methods, and fields specified within the ***Exports*** section of a contract are bound to IUT counterparts. Each binding selection is stored within the contract project so that bindings do not need to be specified each and every time the IUT is run against the contract. Of course, any change in either the IUT or contract will require bindings in the affected area to be re-specified. In addition, a tree like view will be present in Visual Studio 2008 to graphically show the bindings between the contract's structure and the structural elements within the IUT. The contract developer is then able to view, edit, and reset the binding information. The bind also includes the specification of contract parameters that require user specification.

The binding algorithm begins with the first ***MainContract*** and binds the contract to a type within the IUT. Next bindings for observability methods and responsibilities contained within the contract are performed. If an observability or responsibility has a parameter or return type that contains an exported symbol that is not yet bound, binding will be performed for the parameters and return values before the actual observability or responsibility has been bound. Finally, all export lines found within the contract's ***Exports*** section are bound.

When binding non-derived observability methods, an exact parameter and return type match is required. In the case where the requested observability method does not exist within the IUT, the contract developer is able to specify a literal value for the result of an observability instead of an IUT procedure. The rationale for this feature is that in some cases the observability method may not be needed from the IUT's point of view. For an example, consider a BoundedContainer contract. It would be possible to implement a BoundedContainer using an unbounded data structure. As such the BoundedContainer::IsFull() observability method could be hard-wired to yield a false value. When a hard-wired binding is used, the ACL compiler will issue a warning to notify the contract developer of such hard-wiring.

When binding non-derived responsibilities, an actual IUT procedure (or group of procedures) must be specified. However to allow for maximum flexibility responsibilities that do not specify a return type may be bound to an IUT procedure with any return type (including void). Also, when a responsibility specifies a parameter set, that responsibility can be bound to any IUT procedure that has at least the requested parameters. That is, a parameter map will be created between the IUT procedure's parameters and the parameters specified by the responsibility. Any additional parameters specified by the IUT method are simply ignored by the contract. Additional responsibility binding rules have been already discussed within this document and will not be repeated here.

Step 4 – Static Checks

Once binding has been completed, the static checks are executed against the IUT. The IUT is opened using Microsoft Phoenix and a tree structure of the IUT is created internally. The static checks operate against this tree structure. From the implementation point of view, the execution of static checks is fairly straightforward in that the check executes a query against the model of the IUT to obtain a result. The VF's plug-in mechanism allows for custom static checks to be designed and implemented.

Step 5 – Instrumentation

Once the static checks have been completed, the IUT is instrumented in several ways in preparation for scenario execution. The following list indicates how the IUT is instrumented:

- Side-Effect Free Procedures – IUT procedures that have been bound to observability methods are marked so that the profiler can ensure that the state of the system does not change while an observability method is being executed.
- Pre/Post/Inv – IUT procedures that have been bound to responsibilities that contain design-by-contract elements are instrumented so that the corresponding preconditions, post-conditions, and invariants are tested. This also includes instrumentation for the ***PreSet*** construct and the saving of the return value for use in post-conditions. When a design-by-contract construct fails, a special exception will be thrown. The exception will notify the profiler of the construct failure, and will report any beliefs, the stack trace, and the constraint that failed in the contract evaluation report.
- Dynamic Checks – Custom dynamic checks are able to instrument the IUT so that the profiler and the check are able to monitor runtime information. Such checks also include the preservation of metric values recorded within the contract.

It should be noted that the current version of the VF performs less instrumentation than previous versions. The rationale for this is that more information is captured by the runtime and does not need to be explicitly added to the IUT. Additional details regarding instrumentation will be provided once implementation of the VF has been completed. Once instrumentation is complete, the IUT is executed against the profiler to record execution events and requested metrics.

Step 6 – Scenario Evaluation

As the IUT is being executed, the profiler will record when a scenario triggering event occurs. At this point the profiler will create a new scenario instance within the profiler. That is, scenario evaluation is performed on-the-fly. As the profiler is notified of method calls, events, and other activities, any scenario instances that apply to the instance that received the method call/event will be notified of the event. The scenario instance will then determine if the method call/event matches the scenario grammar. Put another way, scenario instance objects can be seen as a Windows application that receives events that occur within their window space. Each scenario instance object will then filter the event to determine if the scenario object should "eat" the event, or if the event has no bearing on the scenario. Such processing involves walking through the scenario grammar, and determining if the scenario termination event occurs. Any scenarios that have yet to terminate when the IUT finishes execution are said to fail.

Step 7 – Non-functional Requirements

Non-functional requirements are checked, through the use of metrics and dynamic checks. Each contract is able to gather metric information as the IUT and corresponding scenarios are executed. Metrics can also be gathered by statically analyzing the IUT. Such static analysis is accomplished through the use of a static check in the Structure section of the contract.

Once the IUT has finished executing each contract's Reports section is processed. The Reports section uses the metric methods defined within the contract to get the required values. These values are then passed to specified metric evaluators to interpret and report on the values. That is, metric evaluators are used to evaluate the metrics gathered while the IUT was executed. The result of such evaluation is reported on the contract evaluation report.

Metric evaluators were chosen as the method for the evaluation of non-functional requirements, due to the subjective nature of non-functional requirements. That is, determining if a given metric is "good" or "bad" depends heavily on the domain in which the IUT exists. As such, contract developers can provide specialized metric evaluators to interpret and report on metrics gathered by the contract framework. These metrics can be gathered through the use of dynamic checks or by static checks analyzing the structure of the IUT.

Step 8 – The Contract Evaluation Report

The Contract Evaluation Report is displayed once the IUT has finished executing and all metrics have been processed. The report contains a summary of the checks performed, and indicates any relations, scenarios, checks, and design-by-contract elements that have failed to execute. The report also contains the results of metric interpretation performed by the metric evaluators. The presentation of the contract evaluation report concludes this example and this document.