ACL Specification

Supports ACL Version 1.0

Dave Arnold
School of Computer Science
Carleton University
Document Version 1.0
10/18/2007

Contents

List of Figures

List of Tables

List of Grammars

1 Introduction

The following document specifies the high-level contract specification language called Another Contract Language (ACL). The ACL is used to specify scenarios and checks that compose a contract. The ACL compiler targets an intermediate language known as the Contract Intermediate Language (CIL). The purpose of the CIL is to allow for multiple, possibly domain specific, high-level contract languages to be used within the contract evaluation framework. The ACL can be seen as a general purpose contract specification language, and as such it is not targeted towards a specific domain.

The document will begin with an overview of the language elements supported by the ACL. These elements will include comments, keywords, tokens, and types. Next, basic concepts required for creating a simple ACL contract are examined in detail. Once the basic concepts have been discussed, each type of section which may reside within a contract is explored. Throughout the document formal grammar sections will be presented. These sections specify the entirety of the ACL.

2 Language Elements

The following sections will examine the various language elements found within the ACL. These are general aspects of the ACL that define comments, white space, literals, keywords, and other elements.

2.1 Comments

ACL comments can be specified using one of two methods. The first is a line comment. Line comments are denoted by using a double forward slash (//). Any text following the double forward slash is considered a comment until the end of the current line is reached. A line comment can begin anywhere on a line. The second method for the specification of a comment is a block comment. Block comments may span multiple lines and begin using the C-style forward slash asterisk notation (/*). Block comments terminate using the reverse notation, an asterisk followed by a forward slash (*/). Grammar 2-1 contains the comment specification.

```
comment:
      line-comment
      block-comment

line-comment:
      //   input-characters_opt

input-characters:
      input-character
      input-characters   input-character

input-character:
      Any Unicode character except a new-line-character

new-line-character:
      Carriage return character (U+000D)
      Line feed character (U+000A)
      Next line character (U+0085)
      Line separator character (U+2028)
      Paragraph separator character (U+2029)

block-comment:
      /*   block-comment-text_opt   asterisks   /

block-comment-text:
      block-comment-section
      block-comment-text   block-comment-section

block-comment-section:
      /
      asterisks_opt   not-slash-or-asterisk

asterisks:
      *
      asterisks   *

not-slash-or-asterisk:
      Any Unicode character except / or *
```

Grammar 2-1 - Comments

As with other programming languages, comments are used to add additional information to the contract specification that will not be used by the ACL compiler. The tokenizer provided by the ACL compiler will remove all comments before constructing the parse tree.

2.2 White Space

White space serves as a delimiter between other language elements and has no other meaning. The most common white space character is the space. Other white space characters include the horizontal tab, vertical tab, form feed, line feed, and carriage return[1]. The amount of white space between language elements has no meaning. Grammar 2-2 defines the notion of whitespace within the ACL.

```
whitespace:
      Any character with Unicode class Zs
      Horizontal tab character (U+0009)
      Vertical tab character (U+000B)
      Form feed character (U+000C)
```

Grammar 2-2 - Whitespace

2.3 Tokens

The ACL contains several types of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not considered tokens; however they act as separators for tokens. The following subsections will outline each token type supported by the ACL.

2.3.1 Identifiers

Identifiers are used to denote aspects of a contract such as a variable or scenario name. The identifier rules correspond to those recommended by the Unicode Standard, with two exceptions. The first is that the underscore character (_) is allowed as an initial character[2]. The second exception is that Unicode escape sequences are not permitted within identifiers. Two identifiers are considered the same if they are identical in name. Identifiers are case sensitive. Grammar 2-3 provides a formal definition for identifiers.

[1] Any Unicode character within the character class Zs is considered as white space.
[2] As is used in the C programming language.

```
identifier:
      An identifier-or-keyword that is not a keyword

keyword: one of
      abstract          atomic            Belief            Boolean
      Character         Check             conforms          context
      Contract          derived           dontcare          Exports
      extends           fail              false             Field
      finalize          exported          Import            Integer
      Invariant         List              MainContract      Metric
      Method            Namespace         new               newInstance
      not               null              Observability     once
      parallel          pass              Post              Pre
      PreSet            Real              refine            Report
      ReportAll         Reports           Responsibility    sameas
      Scenario          String            Structure         Terminate
      Timer             Trigger           true              Type
      Using             Value             value             Void

identifier-or-keyword:
      identifier-start-character identifier-part-characters opt

identifier-start-character:
      letter-character
      _ (U+005F)

identifier-part-characters:
      identifier-part-character
      identifier-part-characters identifier-part-character

identifier-part-character:
      letter-character
      decimal-digit

letter-character: one of
      a  b  c  d  e  f  g  h  I  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x
      y  z  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V
      W  X  Y  Z

decimal-digit: one of
      0  1  2  3  4  5  6  7  8  9
```

Grammar 2-3 - Identifiers

2.3.2 Keywords

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier. The ACL contains 40 core keywords, five modifiers, and nine built-in types. Table 2-1 illustrates the supported ACL keywords, modifiers, and type. A detailed explanation of each item will be provided later in this document. Grammatical support is shown in Grammar 2-3.

Keyword	Definition
	Core Language
abstract	Denotes that the attached element must be refined in a sub-contract
atomic	Denotes the beginning of an atomic scenario block
Belief	Denotes the beginning of a new belief
Check	Defines an assertion
conforms	Indicates that the selected IUT type must conform to the specified contract
context	Denotes the instance of the current contract
Contract	Denotes the beginning of a non-main contract
dontcare	Indicates that the required parameter set is of no importance
Exports	Denotes the beginning of an exports section
extends	Indicates that the current contract extends a previously defined contract
fail	Indicates that the containing precondition, post-condition, invariant, or check will fail
false	Denotes the false literal
finalize	Denotes a responsibility that is evaluated upon instance destruction
Import	Imports extensions developed using the Contract Evaluation Engine Extension SDK
Invariant	Denotes an invariant block within a contract
List	Defines a new variable list
MainContract	Denotes the beginning of a main contract
Metric	Denotes a metric block within a contract
Namespace	Denotes a namespace entry where one or more contracts are located
new	Denotes a responsibility that is evaluated upon instance creation
newInstance	Denotes the creation of a new instance in a scenario block
null	Denotes an empty value
Observability	Denotes an observability block within a contract
parallel	Indicates the beginning of a parallel scenario block
pass	Indicates that the containing precondition, post-condition, invariant, or check will pass
Post	Defines a post-condition
Pre	Defines a precondition
PreSet	Defines an assignment of a value during precondition evaluation to be used in a subsequent post-condition
Report	Reports a metric to the contract evaluation report once for each instance of the contract
ReportAll	Reports a metric to the contract evaluation report once for all instances of the contract
Reports	Denotes the beginning of a metric analysis and reporting block
Responsibility	Denotes a responsibility block within a contract
sameas	Determines if two bound types are bound to the same IUT type
Scenario	Denotes a scenario block within a contract
Structure	Denotes the beginning of a structure block within a contract
Terminate	Defines the termination event(s) for a scenario
Trigger	Defines the triggering event(s) for a scenario
true	Denotes the true literal
Using	Imports contracts located in a specified namespace
Value	Defines a new scalar variable
value	Denotes the values returned by a contract section or the bound method (depending on context of use)

Keyword	Definition
	Modifiers
derived	Indicates that the requested type can also be substituted by a derived type
exported	Indicates that the variable is visible outside of the contract
not	Inverts the value of a Boolean expression
once	Ensures that the specified variable is assigned a value only once
refine	Denotes a contract block that is being refined (extended)
	Built-in Types
Boolean	Indicates a Boolean type
Character	Indicates a character type
Field	Indicates a user defined field (a field within the IUT)
Integer	Indicates an integer type
Method	Indicates a user defined method (a method within the IUT)
Real	Indicates a real number type
String	Indicates a string type
Timer	Defines a new timer variable
Type	Indicates a user defined type (a type within the IUT)
Void	Indicates an empty type

Table 2-1 - ACL keywords

In addition to the keywords, modifiers, and built-in types listed in Table 2-1, specific identifiers can also have a special meaning. Such a special meaning is usually based on its location within the ACL grammar. As an example consider the identifier that follows the ***MainContract*** keyword, this identifier is automatically defined as an exported type, but this does not conflict with using these words as identifiers.

2.3.3 Literals

A literal is a source code representation of a value. The ACL supports Boolean, character, integer, real, string, and null literal values. Grammar 2-4 defines literal support. The following subsections will example each of the literals in detail.

```
literal:
      boolean-literal
      integer-literal
      real-literal
      character-literal
      string-literal
      null-literal
```

Grammar 2-4 - Literals

2.3.3.1 Boolean Literals

Boolean literals are used to specify true/false values. A Boolean literal is specified by either using the ***true*** or ***false*** ACL keyword as defined in Grammar 2-4. In addition to the true and false literal values the ***not*** modifier can also be used to invert the value of a Boolean expression. That is, ***not true*** is the same as specifying the ***false*** keyword.

```
boolean-literal:
      true
      false
```

Grammar 2-5 - Boolean literals

2.3.3.2 Character Literals

A character literal represents a single character, and usually consists of a single character surrounded by single quotes, as in 'a'. Grammar 2-6 defines a character literal. The escape sequences illustrated in Table 2 also apply to character literals. That is, '\'' and '\\' are valid character literals. For readers familiar with the C language, character literals in the ACL behave exactly the same.

```
character-literal:
      ‘ character ‘

character:
      single-character
      simple-escape-sequence
      unicode-escape-sequence

single-character:
      Any character except ‘ (U+0027), \ (U+005C), and new-line-character

new-line-character:
      Carriage return character (U+000D)
      Line feed character (U+000A)
      Next line character (U+0085)
      Line separator character (U+2028)
      Paragraph separator character (U+2029)

simple-escape-sequence: one of
      \’  \”  \\  \0  \a  \b  \f  \n  \r  \t  \v

unicode-escape-sequence:
      \u hex-digit hex-digit hex-digit hex-digit
      \U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
            hex-digit hex-digit

hex-digit: one of
      0  1  2  3  4  5  6  7  8  9  0  a  b  c  d  e  f  A  B  C  D  E  F
```

Grammar 2-6 - Character literals

Escape Sequence	Character Name	Unicode Encoding
\'	Single quote	0x0027
\"	Double quote	0x0022
****	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

Table 2-2 - Escape sequences

2.3.3.3 Integer Literals

Integer literals are used to specify integral values. All integral values must be specified in decimal format (base 10). An integer consists of a series of one or more decimal digits, as shown in Grammar 2-7. Unlike other high-level programming languages, such as C or Java, the ACL does not support different integral types. In ACL all integer literals are of the same type. ACL does not support integer literals that fall out of the following range: -2 147 483 648 (-2^{31}) to 2 147 483 648 (2^{31}).

```
integer-literal:
      decimal-digits

decimal-digits:
      decimal-digit
      decimal-digits decimal-digit

decimal-digit: one of
      0  1  2  3  4  5  6  7  8  9
```

Grammar 2-7 - Integer literals

2.3.3.4 Real Literals

Real literals are used to specify floating point values. Real literals can be specified by using zero or more decimal digits, a dot (.), and one or more additional digits. Real literals can also be specified by using the 'e' notation. Format specification for real literals can be found in Grammar 2-8.

```
real-literal:
      decimal-digits . decimal-digits exponent-part_opt
      . decimal-digits exponent-part_opt
      decimal-digits exponent-part

decimal-digits:
      decimal-digit
      decimal-digits decimal-digit

decimal-digit: one of
      0  1  2  3  4  5  6  7  8  9

exponent-part:
      e sign_opt decimal-digits
      E sign_opt decimal-digits

sign:
      -
      +
```

Grammar 2-8 - Real literals

Real literals have seven digit precision and must not fall out of the following range: $\pm 1.5^{e-45}$ to $\pm 3.4^{e38}$. Each of the following literals are legal real literals:

1) 1.5
2) 123.456
3) 0.23
4) 1^{e10}
5) -4.455^{e-4}

2.3.3.5 String Literals

A string literal consists of a double-quote (“) character, zero or more characters, and a closing double-quote (“) character. The characters between the delimiters are interpreted verbatim, with the only exception being an escape sequence. An escape sequence is defined as two characters that are translated into non-printable characters. The first character of an escape sequence is always the backslash character (\) followed by one of the following characters: ‘, “, \, 0, a, b, f, n, r, t, or v. Table 2-2 illustrates the supported escape characters and their meaning. A formal definition for string literals can be found in Grammar 2-9.

```
string-literal:
      “ string-literal-characters_opt “

string-literal-characters:
      string-literal-character
      string-literal-characters string-literal-character

string-literal-character:
      single-string-character
      simple-escape-sequence
      unicode-escape-sequence

single-string-character:
      Any character except “ (U+0022), \ (U+005C), and new-line-character

new-line-character:
      Carriage return character (U+000D)
      Line feed character (U+000A)
      Next line character (U+0085)
      Line separator character (U+2028)
      Paragraph separator character (U+2029)

simple-escape-sequence: one of
      \’  \”  \\  \0  \a  \b  \f  \n  \r  \t  \v

unicode-escape-sequence:
      \u hex-digit hex-digit hex-digit hex-digit
      \U hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
            hex-digit hex-digit

hex-digit: one of
      0  1  2  3  4  5  6  7  8  9  0  a  b  c  d  e  f  A  B  C  D  E  F
```

Grammar 2-9 - String literals

2.3.3.6 Null Literals

The null literal is specified using the ***null*** keyword. A null literal specifies an empty or undefined value. Grammar 2-10 defines the null literal.

```
null-literal:
      null
```

Grammar 2-10 - Null literals

2.3.4 Operators

The ACL supports a number of operators. Operators are used in expressions to describe operations using one or more operands. Table 2-3 illustrates the supported ACL operators and their meaning. Some of the operators in Table 2-3 appear more than once, such occurrence indicates that the operator has a different meaning depending on the context in which the operator is used. Details on operator usage will be presented later.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***
+	Addition, in ***c = a + b***, ***c*** is assigned the result of adding ***a*** and ***b*** together
-	Subtraction, in ***c = a – b***, ***c*** is assigned the result of subtracting ***b*** from ***a***
*	Multiplication, in ***c = a * b***, ***c*** is assigned the result of multiplying ***a*** by ***b***
/	Division, in ***c = a / b***, ***c*** is assigned the result of dividing ***b*** into ***a***
-	Negation, in ***a*** = ***–b***, ***a*** is assigned the result of negating ***b***
+	Identity[3], in ***a = +b***, ***a*** is assigned the value stored in ***b***
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise
>	Greater than, in ***a > b***, the expression returns true if ***a*** is greater than ***b*** and false otherwise
<	Less than, in ***a < b***, the expression returns true if ***a*** is less than ***b*** and false otherwise
>=	Greater than or equal, in ***a >= b***, the expression returns true if ***a*** is greater or equal to ***b*** and false otherwise
<=	Less than or equal, in ***a <= b***, the expression returns true if ***a*** is less or equal to ***b*** and false otherwise
sameas	Type binding equality, in ***a sameas b***, the expression returns true if ***a*** is bound to the same IUT type as ***b***
	Logical Operators
||	Or, in ***a || b***, the expression returns true if ***a*** or ***b*** or both ***a*** and ***b*** evaluate to true, false otherwise
&&	And, in ***a && b***, the expression returns true if both ***a*** and ***b*** evaluate to true, false otherwise
not	Not, in ***not a***, the expression returns true if ***a*** is false, and false otherwise

[3] The unary + operator is included for language completeness.

Operator	Definition
	Scenario Operators
,	Follows, in ***a, b***, event ***a*** occurs followed by event ***b***
*	Zero or more, in ***a****, event ***a*** occurs zero or more times
+	One or more, in ***a+***, event ***a*** occurs one or more times
[n]	Exactly n, in ***a[n]***, event ***a*** occurs exactly n times
|	Or, in ***a | b***, event ***a*** or event ***b*** can occur
&	And, in ***a & b***, event ***a*** and event ***b*** must both occur in any order
;	End, in ***a;***, event ***a*** indicates the end of the scenario grammar
atomic	Atomic action, in ***atomic { a, b }***, event ***a*** followed by event ***b*** can be seen as a single atomic action from the viewpoint of the scenario
parallel	Parallel action, in ***parallel {a, b }***, event ***a*** followed by event ***b*** can contain multiple active scenario instances
	Other Operators
{	Open scope
}	Close scope
()	Denotes a method call with an optional parameter list
~	Current contract identifier (used in beliefs only)
.	Member access operator
::	Scope resolution operator (used in binding rules only)

Table 2-3 - ACL operators

2.3.5 Operator Precedence and Associativity

When an expression contains multiple operators, the precedence of the operators controls the order that the individual operators are evaluated. For example, the expression ***a + b * c*** is evaluated as ***a + (b * c)*** because the ******* operator has higher precedence than the binary ***+*** operator. The precedence of an operator is established by the definition of its associated grammar production.

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order that the operations are performed. Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed left to right. The assignment operators are right-associative, meaning that operations are performed from right to left. Precedence and associativity can be controlled using parentheses. Table 2-4 summarizes all of the ACL non-scenario operators in order of precedence from highest to lowest. Table 2-5 summarizes all of the ACL scenario operators in order of precedence from highest to lowest.

Operator	Category
() .	Primary
- + not	Unary
* /	Multiplicative
+ -	Additive
< > <= >=	Relational
== sameas	Equality
&&	Conditional AND
||	Conditional OR
=	Assignment

Table 2-4 - Operator precedence

Operator	Category
+ * [n]	Repetition
&	Logical AND
|	Logical OR
parallel atomic	Temporal
,	Follows

Table 2-5 - Scenario operator precedence

2.3.6 Punctuators

The ACL supports a number of punctuators. Punctuators are used within contracts for ordering and grouping of elements. Table 2-6 illustrates the supported ACL punctuators and their meaning. Details on punctuator usage will be presented later in this document.

Punctuator	Definition
,	Parameter list separator
;	Statement terminator
(	Denotes the beginning of an increased precedence section
)	Denotes the ending of an increased precedence section
<	Denotes the beginning of a generic parameter list
>	Denotes the ending of a generic parameter list

Table 2-6 - ACL punctuators

2.3.7 Summary

The previous sections have illustrated the various types of tokens that are supported under the ACL. We will now examine the type system used by the ACL, and the corresponding built-in types.

2.4 Types

The ACL does not contain the notion of types as defined in other high-level programming languages such as C++. The ACL uses identifiers that are mapped to actual types defined within the IUT. To this end, there are only 10 built-in types. In addition, every type that is defined within the IUT is available for use within the ACL. The ACL uses identifiers that are bound to the actual types, via a binding tool, to represent types. The binding activities are performed via the framework's binding tool. Such binding includes both user defined types, and the built-in types that exist in the programming language used to create the IUT. For example, in C# there is an ***int*** type. In the ACL, an identifier named ***int*** can be defined to map to the C# ***int*** type.[4] Two types are considered equal if they are bound to the same type in the IUT, regardless of their identifier names.

The 10 built-in ACL types are provided to aid with the creation of contracts. The built-in types are listed in Table 1. The following subsections will examine each of the built-in types.

2.4.1 The Boolean Type

The ***Boolean*** type is used to represent a Boolean literal value. Variables defined using the ***Boolean*** type either have a ***true*** or ***false*** value. Variables of the ***Boolean*** type must be initialized to either a ***true*** or ***false*** value before use. If an uninitialized variable is used as an r-value a compile-time error will be generated. The operators shown in Table 2-7 can be used with the ***Boolean*** type. In all cases the variable ***a*** is of the ***Boolean*** type. The variable ***b*** must be either a Boolean literal, a variable of the ***Boolean*** type, or an expression that yields a Boolean value.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***.
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise.
	Logical Operators
||	Or, in ***a || b***, the expression returns true if ***a*** or ***b*** or both ***a*** and ***b*** evaluate to true, the expression returns false otherwise.
&&	And, in ***a && b***, the expression returns true if both ***a*** and ***b*** evaluate to true, false the expression returns false otherwise.
not	Not, in ***not a***, the expression returns true if ***a*** is false, and false otherwise.

Table 2-7 - Operators for the Boolean type

[4] The C# int type is defined by the System.Int32 class.

2.4.2 The Character Type

The ***Character*** type is used to represent a character literal. Variables of the ***Character*** type must be assigned a value before use. That is, the variable cannot be used as an r-value before it has been assigned a value. If such a use occurs, a compile-time error will be generated. The operators shown in Table 2-8 can be used with the ***Character*** type. In all cases the variable ***a*** is of the ***Character*** type. The variables ***b*** and ***c*** must be either a character literal, a variable of the ***Character*** type, or an expression which yields a ***Character*** value.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***.
+	Addition, in a ***= b + c***, ***a*** is assigned the result of adding ***b*** and ***c*** together. The addition is performed by taking the Unicode value of ***b*** and ***c*** and adding them together to get the Unicode value for the character assigned to ***a***. For example, '1' + '1' will result in the 'b' character. This is because the Unicode value of '1' is 0x0031, which will yield a value of 0x0062 after the addition, creating the 'b' character. In ***a = b + d***, where ***d*** is either an integer literal, a variable of the ***Integer*** type, or an expression which yields an ***Integer*** value, ***a*** is assigned the result of adding ***b*** and ***d*** together. The addition is performed by taking the Unicode value of ***b*** and adding ***d*** to get the Unicode value for the character assigned to ***a***. For example, 'A' + 1 will result in the 'B' character. If any character addition results in an overflow, a run-time error will be generated.
-	Subtraction, in ***a = b – c***, a is assigned the result of subtracting c from b. The subtraction is performed by taking the Unicode values of ***b*** and ***c*** and subtracting ***c*** from ***b*** to get the Unicode value for the character assigned to ***a***. For example, 'n' – '!' will result in the 'M' character. This is because the Unicode value of 'n' is 0x006E and the Unicode value of '!' is 0x0021, which will yield a value of 0x004D after the addition, creating the 'M' character. In ***a = b - d***, where ***d*** is either an integer literal, a variable of the ***Integer*** type, or an expression which yields an ***Integer*** value, ***a*** is assigned the result of subtracting ***d*** from ***b***. The subtraction is performed by taking the Unicode value of ***b*** and subtracting ***d*** to get the Unicode value for the character assigned to ***a***. For example, 'B' - 1 will result in the 'A' character. If any character subtraction results in an underflow, a run-time error will be generated.

Operator	Definition
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise. Two ***Character*** types are considered equal if and only if, they have the same Unicode values.
>	Greater than, in ***a > b***, the expression returns true if ***a*** is greater than ***b*** and false otherwise. ***a*** is considered greater than ***b*** if and only if ***a***'s Unicode value is larger than ***b***'s Unicode value.
<	Less than, in ***a < b***, the expression returns true if ***a*** is less than ***b*** and false otherwise. ***a*** is considered less than ***b*** if and only if ***a***'s Unicode value is less than ***b***'s Unicode value.
>=	Greater than or equal, in ***a >= b***, the expression returns true if ***a*** is greater or equal to ***b*** and false otherwise. ***a*** is considered greater than or equal to ***b*** if and only if ***a***'s Unicode value is larger or equal to ***b***'s Unicode value.
<=	Less than or equal, in ***a <= b***, the expression returns true if ***a*** is less or equal to ***b*** and false otherwise. ***a*** is considered less than or equal to ***b*** if and only if ***a***'s Unicode value is less or equal to ***b***'s Unicode value.

Table 2-8 - Operators for the Character type

2.4.3 The Field Type

The ***Field*** type is used to map an identifier to a field contained within the IUT. Fields cannot be created or assigned to, as they have to be bound to an existing field within the IUT. The binding operation is performed by the framework's binding tool. The ***Field*** type only supports a single operator, binding equality. Table 2-9 illustrates the supported operator. In all cases ***a*** and ***b*** are either variables of the ***Field*** type, or an expression which yields a ***Field*** value.

Operator	Definition
	Comparison Operators
sameas	Binding equality, in ***a sameas b***, the expression returns true if ***a*** is bound to the same type as ***b*** and false otherwise. Two ***Field*** types are considered the same if they are bound to the same IUT field.

Table 2-9 - Operators for the Field type

2.4.4 The Integer Type

The ***Integer*** type represents an integer literal value. Variables defined using the ***Integer*** type have a value that corresponds to the previously discussed integer literals. Variables of the ***Integer*** type must be initialized to an integral value before use. Use of an uninitialized integer variable in a r-value expression will result in a compile-time error. The operators shown in Table 2-10 can be used with the ***Integer*** type. In all cases the variable ***a*** is of the ***Integer*** type. The variables ***b*** and ***c*** must either be an integer literal, a variable of the ***Integer*** type, or an expression which yields an integer value.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***.
+	Addition, in a ***= b + c***, ***a*** is assigned the result of adding ***b*** and ***c*** together. In ***a = b + d***, where ***d*** is either a character literal, a variable of the ***Character*** type, or an expression which yields a character value, ***a*** is assigned the result of adding ***b*** and ***d*** together. The addition is performed by converting the character, ***d***, into its base-10 Unicode value and adding that value to ***b***. For example the addition of 2 + 'A' yields a value of 67. This is because the Unicode value for 'A' is 0x0041, which is 65 in base-10, so the expression becomes 2 + 65, yielding a result of 67. If an addition yields a result which falls outside of the integer literal range, a run-time error will be generated.
-	Subtraction, in ***a = b – c***, ***a*** is assigned the result of subtracting ***c*** from ***b***. In ***a = b – d***, where ***d*** is either a character literal, a variable of the ***Character*** type, or an expression which yields a character value, ***a*** is assigned the result of subtracting ***d*** from ***b***. The subtraction is performed by converting the character, ***d***, into its base-10 Unicode value and subtracting that value from ***b***. For example the result of 2 – '$' yields a value of -34. This is because the Unicode value for '$' is 0x0024, which is 36 in base-10, so the expression becomes 2 – 36, yielding a result of -34. If a subtraction yields a result which falls outside of the integer literal range, a run-time error will be generated.
*	Multiplication, in ***a = b * c***, ***a*** is assigned the result of multiplying ***b*** by ***c***. If a multiplication yields a result that falls outside of the integer literal range, a run-time error will be generated.
/	Division, in ***a = b / c***, ***a*** is assigned the result of dividing ***c*** into ***b***. As ***a*** is an integer type, only whole divisions will be returned. That is, any fraction or remainder created as a result of the division will be lost. For example, the result of 5 / 6 is 0, and the result of 13 / 6 is 2.
-	Negation, in ***a*** = ***–b***, ***a*** is assigned the result of negating ***b***.
+	Identity, in ***a = +b***, ***a*** is assigned the value stored in ***b***.

Operator	Definition
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise. Two ***Integer*** types are equal if they represent the same integer literal.
>	Greater than, in ***a > b***, the expression returns true if ***a*** is greater than ***b*** and false otherwise. ***a*** is considered greater than ***b*** if and only if ***a***'s integer literal is larger than ***b***'s integer literal value.
<	Less than, in ***a < b***, the expression returns true if ***a*** is less than ***b*** and false otherwise. ***a*** is considered less than ***b*** if and only if ***a***'s integer literal is smaller than ***b***'s integer literal value.
>=	Greater than or equal, in ***a >= b***, the expression returns true if ***a*** is greater or equal to ***b*** and false otherwise. ***a*** is considered greater than or equal to ***b*** if and only if ***a***'s integer literal is larger or equal to ***b***'s integer literal value.
<=	Less than or equal, in ***a <= b***, the expression returns true if ***a*** is less or equal to ***b*** and false otherwise. ***a*** is considered less than or equal to ***b*** if and only if ***a***'s integer literal is smaller than or equal to ***b***'s integer literal value.

Table 2-10 - Operators for the Integer type

2.4.5 The Method Type

The ***Method*** type is used to map an identifier to a method contained within the IUT. Methods cannot be created or assigned to, as they have to be bound to an existing method within the IUT. The binding operation is performed by the framework's binding tool. The ***Method*** type only supports a single operator, binding equality. Table 2-11 illustrates the supported operator. In all cases ***a*** and ***b*** are either variables of the ***Method*** type, or an expression which yields a ***Method*** value.

Operator	Definition
	Comparison Operators
sameas	Binding equality, in ***a sameas b***, the expression returns true if ***a*** is bound to the same type as ***b*** and false otherwise. Two ***Method*** types are considered the same if they are bound to the same IUT method. It should be noted that the method's complete signature, return type, name, and parameter types are used for the comparison.

Table 2-11 - Operators for the Method type

2.4.6 The Real Type

The ***Real*** type represents a real literal value. Variables defined using the ***Real*** type have a value that corresponds to the previously discussed real literals. Variables of the ***Real*** type must be initialized to a real value before use. Use of an uninitialized real variable in a r-value expression will result in a compile-time error. The operators shown in Table 2-12 can be used with the ***Real*** type. In all cases the variable ***a*** is of the ***Real*** type. The variables ***b*** and ***c*** must either be a real literal, a variable of the ***Real*** type, or an expression which yields a real value.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***.
+	Addition, in a ***= b + c***, ***a*** is assigned the result of adding ***b*** and ***c*** together. In ***a = b + d***, where ***d*** is either an integer literal, a variable of the ***Integer*** type, or an expression which yields a real value, ***a*** is assigned the result of adding ***b*** and ***d*** together. The addition is performed by converting the integer, ***d***, into a real number and adding that value to ***b***. For example the addition of 3.14 + 4 yields a value of 7.14. This is because the integer value 4, is converted to the real value 4.0, so the expression becomes 3.14 + 4.0, yielding a value of 7.14. If an addition yields a result that falls outside of the real literal range, a run-time error will be generated.
-	Subtraction, in ***a = b – c***, ***a*** is assigned the result of subtracting ***c*** from ***b***. In ***a = b – d***, where ***d*** is either an integer literal, a variable of the ***Integer*** type, or an expression which yields an integer value, ***a*** is assigned the result of subtracting ***d*** from ***b***. The subtraction is performed by converting the integer, ***d***, into a real number to subtract from ***b***. For example the subtraction of 88.53 – 88 yields a value of 0.53. This is because the integer value 88, is converted to the real value 88.0, so the expression becomes 88.53 - 88, yielding a value of 0.53. If a subtraction yields a result that falls outside of the real literal range, a run-time error will be generated.
*	Multiplication, in ***a = b * c***, ***a*** is assigned the result of multiplying ***b*** by ***c***. If a multiplication yields a result that falls outside of the real literal range, a run-time error will be generated.
/	Division, in ***a = b / c***, ***a*** is assigned the result of dividing ***c*** into ***b***.
-	Negation, in ***a*** = ***–b***, ***a*** is assigned the result of negating ***b***.
+	Identity, in ***a = +b***, ***a*** is assigned the value stored in ***b***.

Operator	Definition
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise. Two ***Real*** types are equal if they represent the same real literal.
>	Greater than, in ***a > b***, the expression returns true if ***a*** is greater than ***b*** and false otherwise. ***a*** is considered greater than ***b*** if and only if ***a***'s real literal is larger than ***b***'s real literal value.
<	Less than, in ***a < b***, the expression returns true if ***a*** is less than ***b*** and false otherwise. ***a*** is considered less than ***b*** if and only if ***a***'s real literal is smaller than ***b***'s real literal value.
>=	Greater than or equal, in ***a >= b***, the expression returns true if ***a*** is greater or equal to ***b*** and false otherwise. ***a*** is considered greater than or equal to ***b*** if and only if ***a***'s real literal is larger or equal to ***b***'s real literal value.
<=	Less than or equal, in ***a <= b***, the expression returns true if ***a*** is less or equal to ***b*** and false otherwise. ***a*** is considered less than or equal to ***b*** if and only if ***a***'s real literal is smaller than or equal to ***b***'s real literal value.

Table 2-12 - Operators for the Real type

2.4.7 The String Type

The ***String*** type represents a string literal value. Variables defined using the ***String*** type have a value that corresponds to the previously discussed string literals. Strings must be initialized to a value before use. Use of an unassigned string variable in a r-value expression will result in a compile-time error. The operators shown in Table 2-13 can be used with the ***String*** type. In all cases the variable ***a*** is of the ***String*** type. The variables ***b*** and ***c*** must either be a string literal, a variable of the ***String*** type, and an expression that yields a string value.

Operator	Definition
	Mathematical Operators
=	Assignment, in ***a = b***, ***a*** is assigned the value of ***b***.
+	Concatenation, in ***a = b + c***, ***a*** is assigned the result of concatenating ***b*** followed by ***c***.
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise. Two ***String*** types are considered equal if and only if they contain string literals of the same length, and the characters of each string are equal.

Table 2-13 - Operators for the String type

2.4.8 The Timer Type

The ***Timer*** type is used to represent a run-time timer. Timers operate by recording the number of application "ticks" that have elapsed since the timer was started. Real time is not used, as our framework operates via Microsoft's .NET run-time. As such application ticks are used to represent time. Timers are used to gather metric information that can be used by metric evaluators for the evaluation of non-functional requirements. Each instance of the ***Timer*** type can simultaneously maintain an unlimited number of timers.[5] Figure 2-1 illustrates an example timer declaration. The ***Timer*** type does not define any operators. That is, the use of any operator on a timer object will result in a compile-time error. The ***Timer*** type does define three operations which can be invoked to start, stop, and query the value of a timer. The following subsections will present the three operations defined on the ***Timer*** type.

```
Timer item_timer;
```

Figure 2-1 - Timer declaration

2.4.8.1 Start(x)

The ***Start(x)*** operation is used to initialize and start a new timer. The ***Start(x)*** operation accepts a single parameter that acts as a key for the timer. There is no specific type for the parameter. That is, you could assign a timer to a ***Customer*** type that is defined within the IUT, or an integer literal could be used. The only restriction is that a ***null*** value cannot be used. If a ***null*** value is used, a run-time error will be generated. If a timer has already been initialized for the given parameter, the existing timer will be reinitialized.

2.4.8.2 Stop(x)

The ***Stop(x)*** operation is used to stop a timer that has been initialized via the ***Start(x)*** operation. The single parameter required by the ***Stop(x)*** operation must be the same as the one provided to ***Start(x)***. If no timer has been started for the given parameter, a run-time error will be generated. If the ***Stop(x)*** operation is invoked on a timer that is not currently running, a run-time error will also be generated. Once a timer has been stopped, the number of application ticks that have elapsed while the timer was running is stored and can be accessed via the ***Value(x)*** operation.

2.4.8.3 Value(x)

The ***Value(x)*** operation is used to retrieve the number of application ticks that have elapsed since the timer was started, in the case where the timer is still running, or the number of ticks which have elapsed between the invocations of the ***Start(x)*** and ***Stop(x)*** operations. The single parameter passed to the ***Value(x)*** operation must be the same as the one provided to the ***Start(x)*** operation. If no timer exists for the given parameter, a run-time error will be generated.

2.4.8.4 Summary

The ***Timer*** built-in type provides a method for recording the number of application ticks that have elapsed between two events. The operations described in the previous sections provide the ability to start, stop, and retrieve the value of the timer.

[5] Constrained by system resources.

2.4.9 The Type Type

The ***Type*** type is used to map an identifier to a type contained within the IUT. Types cannot be created or assigned to, as they have to be bound to an existing type within the IUT. The binding operation is performed by the framework's binding tool. The ***Type*** type only supports a single operator, binding equality. Table 2-14 illustrates the supported operator. In all cases ***a*** and ***b*** are either variable of the ***Type*** type, or an expression which yields a ***Type*** value.

Operator	Definition
	Comparison Operators
sameas	Binding equality, in ***a sameas b***, the expression returns true if ***a*** is bound to the same type as ***b*** and false otherwise. Two ***Field*** types are considered the same if they are bound to the same IUT field.

Table 2-14 - Operators for the Type type

2.4.10 The Void Type

The ***Void*** type is used to specify a return type for an observability method, responsibility, or scenario that does not return a value. The ***Void*** type does not need to be explicitly specified in ACL. A lack of return type specification will always be interpreted as the ***Void*** type. The ***Void*** type cannot be used to declare variables. As such, the ***Void*** type does not support any operators, or contain any predefined operations.

2.4.11 Summary

The ACL contains a unique type system that allows types, methods, and fields that are defined within the IUT to be used. The ACL defines a set of primitive types that can be used to store Boolean, character, integer, real, and string values. The ACL also contains a timer as a built-in type for calculating metric values. Each of the types described in this section can be used as a scalar or in a list. The following section will differentiate between the two types, and describe list operations.

2.5 Value Types and List Types

The ACL supports two types of variable declarations, scalar and list. The scalar type of variable is used to specify a single instance. The list type of variable creates a strongly typed, ordered, and unbounded collection. The following two subsections will provide details of each variable declaration type.

2.5.1 The Value Type

The value type is denoted by the ***Value*** keyword. As illustrated in Figure 2-2 the ***Value*** keyword is placed before the type of variable to be created. As the name suggests, a value type represents a variable of a specified type that holds a single value.

```
Value Integer aNumber;
```

Figure 2-2 - Use of the Value keyword

The value type is considered the default variable type, and as such the use of the ***Value*** keyword is not mandatory. In all cases, when the variable declaration type is missing, the value type is assumed. Variables declared using the value type can also be preceded by one or more modifiers. Modifiers will be discussed in the next section.

2.5.2 The List Type

As the name suggests, the list type represents a built-in generic collection object. Lists are unbounded, and ordered. As Figure 2-3 illustrates, the ***List*** keyword is placed before the variable type to be declared. As the list type is not the default, the use of the ***List*** keyword is mandatory. As the ***List*** keyword is immediately followed by the variable type, all lists only contain a single type of element. That is, lists are strongly typed.

```
List Integer customer_times;
```

Figure 2-3 - List declaration

Lists only support a single operator, equality. The list equality operator can only be applied to lists that are derived from the same variable type. If the list equality operator is applied to two differently typed lists a compile-time error will be generated. Table 2-15 illustrates the supported operator. In all cases ***a*** and ***b*** are lists of the same variable type, or an expression which yields a list value.

Operator	Definition
	Comparison Operators
==	Equality, in ***a == b***, the expression returns true if ***a*** is equal to ***b*** and false otherwise. Two lists are considered equal if and only if each list contains the same elements in the same order.

Table 2-15 - List operators

Variables defined as a list contain a series of predefined operations that can be used to add, remove, and inspect elements. The following subsections will explore each of these operations.

2.5.2.1 Add(x)

The ***Add(x)*** operation inserts the given element, x, into the list. If the element has already been added to the list, an additional element is added. That is, a list may contain duplicate elements. The type of element that is added to the list must be identical to the type used when declaring the variable. In the case of Figure 2-3, only integers can be added to the ***customer_times*** list. A compile-time error will be issued if an attempt is made to add a non-integer type to the ***customer_times*** list. Elements are always added to the end of the list. Lists do not contain an upper bound. That is, a list can theoretically contain an unlimited number of items.[6]

2.5.2.2 At(index)

The ***At(index)*** operation retrieves an element at the specified index. The index parameter is of the built-in ***Integer*** type. The element at the specified index is returned to the caller. Lists are zero based. The return type of the ***At(index)*** operation is the list's element type. If the specified index does not reference a valid element, a run-time error will be generated. The ***At(index)*** operation is the recommended method for accessing elements within the list, without removal.

2.5.2.3 Contains(x)

The ***Contains(x)*** operation is used to determine if one or more instances of the given element is stored within the list. If the given element is present, ***Contains(x)*** returns ***true***. If the given element is not stored within the list, ***Contains(x)*** returns ***false***. The element passed to ***Contains(x)*** must be of the same type used in the list's declaration. If not a compile-time error will be generated.

2.5.2.4 Count(x)

Count(x) is used to determine how many instances of the given element are stored within the list. If no elements are stored in the list ***Count(x)*** will return zero. As with other list operations, the element passed to the ***Count(x)*** operation must be of the same type used in the list's declaration.

2.5.2.5 IndexOf(x)

The ***IndexOf(x)*** operation determines the zero-based index of the first occurrence of the specified element. If there are no instances of the given element within the list, a negative value will be returned. The element passed to ***IndexOf(x)*** must be of the same type used in the list's declaration; otherwise a compile-time error will be generated.

2.5.2.6 Init()

The ***Init()*** operation is used to initialize the list and to clear any existing elements. The ***Init()*** operation must be invoked prior to using any other list operations. If the ***Init()*** operation is not invoked, a run-time error will be issued. ***Init()*** can also be used to return a list to its initial condition. There is no limit on the number of times that ***Init()*** can be invoked.

2.5.2.7 Insert(index, x)

Insert(index, x) inserts the specified element, x, at the location specified by the index parameter. The element to be inserted must be of the same type used during the list's declaration. The

[6] The maximum number of elements that can be contained within a set is defined by the size required to store the element type, and the amount of available system memory.

specified index must be within the boundaries of the list. That is, an element cannot be inserted beyond the end of the list. The index is zero based, and the element to insert is placed at that location, the remainder of the list is pushed down to accommodate the new element.

2.5.2.8 Length()

The ***Length()*** operation returns an integer value indicating the number of elements in the list. The length of the list is the number of elements currently being stored within the list. If two instances of the same element are stored within the list, ***Length()*** will return two. That is, each copy of an element contained within the list is counted. If the list is empty ***Length()*** will return a value of zero.

2.5.2.9 Remove(x)

Analogous to the ***Add(x)*** operation, ***Remove(x)*** removes the first instance of the element, x, found within the list. ***Remove(x)*** begins at the front the list and moves down until a matching element is encountered. If no such element exists within the list, ***Remove(x)*** performs no action. The type of element which is passed to ***Remove(x)*** must be identical to the type used in the list's declaration.

2.5.2.10 RemoveAll(x)

RemoveAll(x) is similar to the ***Remove(x)*** operation, except all instances of the given element are removed from the list. If no instances of the given element exist within the list, ***RemoveAll(x)*** performs no action. The same typing rules that apply to ***Remove(x)***, also apply to ***RemoveAll(x)***.

2.5.2.11 RemoveAt(index)

The ***RemoveAt(index)*** operation removes the element located at the specified zero-based index. If the specified index does not contain an element, a run-time error will be generated. Once the requested element has been removed from the list, all of the elements that were located further down the list are shifted up to close the gap left from the element's removal.

2.5.2.12 RemoveFirst()

The ***RemoveFirst()*** operation removes the first element contained within the list. The element is returned to the caller. That is, the element located at index zero is removed from the list and returned to the caller. The return type of ***RemoveFirst()*** is the type used in the list's declaration. In the case of Figure 2-3 the return type would be the ***Integer*** built-in type. In the case where there are no elements in the list, a call to ***RemoveFirst()*** will result in a run-time error.

2.5.2.13 Summary

The list variable type represents a strongly typed, unbounded, and ordered collection of elements. The previous sections have presented operations for the addition, removal, and inspection of elements. We will now examine variable modifiers that can be used to specialize a given contract variable.

2.6 Variable Modifiers

When declaring variables using the two previous variable types, and the previously discussed built-in types, modifiers may be used to provide additional semantics for the variable being declared. Variable modifiers are placed before the variable type. That is, variable modifiers are the first element specified in a variable declaration. There are two modifiers that apply to variables. The following subsections will present them.

2.6.1 The Exported Modifier

The exported modifier is denoted by the ***exported*** keyword. As the name suggests, exported variables are visible outside the contract of definition. That is, the variable's value can be accessed by other contracts. The exported modifier has the same behaviour as the ***public*** access modifier found in C++. If the exported modifier is not specified the variable declaration is considered to be protected, in the C++ sense, and cannot be viewed outside of the contract where the variable is defined. Variables that are defined with or without the exported modifier in a contract that is inherited are visible by all sub-contracts, regardless of inheritance depth. Examples regarding the use of the exported modifier will be presented shortly.

2.6.2 The Once Modifier

The once modifier, denoted by the ***once*** keyword, indicates that the variable being declared can only be assigned to once. That is, the variable can only be used as an l-value a single time. Experience using the ACL language to implement contracts has shown that a large number of contract variables are only assigned a value once, and then that value is queried several times. In order to support and enforce such a pattern, the once modifier was developed. If a variable defined with the once modifier is assigned a value more than once a compile-time error will be generated.

2.6.3 Summary

Variable modifiers are used to add additional semantic information to a variable being declared. The modifiers discussed in the previous sections allow for exporting a variable from a contract and for ensuring that a variable is only assigned to once. With the basic language discussion complete, we will now examine the ACL extension mechanism.

3 Plug-ins

In order to provide an open set of checks and evaluators that can be performed on an IUT, the ACL supports language plug-ins. These plug-ins are user specified static checks, run-time checks, and metric evaluators. Such checks and evaluators allow additional user-defined functionality to be integrated within the ACL. Plug-in creation is targeted towards plug-in developers, rather than contract writer. Additional information regarding the creation of plug-ins is provided in a separate technical report. For the purposes of the ACL specification, we will define how such plug-ins can be referenced and used within a contract specified in ACL.

3.1 Plug-in Types

The ACL supports three types of plug-ins: static checks, run-time checks, and metric evaluators. The following provides a brief overview of each plug-in type, and how it can be used within the ACL contract.

3.1.1 Static Checks

As the name suggests, static checks perform a check on the IUT that can be accomplished without execution. Examples of static checks include: analysis of field access modifiers, tests on inheritance depth, and the structural use of design patterns. Static checks can only be called by placing code in the structure section of a contract. Details on the structure section will be presented shortly. Static checks cannot be called from any other section of the contract. A static check can be viewed as an operation. Each check has a return type and any number of parameters. A static check is guaranteed to be side-effect free.

3.1.2 Run-time Checks

A run-time check is used to perform a test on the IUT during execution. That is, the check can only be evaluated while the IUT is being executed. Examples of run-time checks include: testing a value of a variable at a given point, ensuring a given state exists within an object, and validating data sent between two different objects. Run-time checks can be invoked by placing code in the observability, responsibility, or scenario sections of a contract. Details on all contract sections will be presented shortly. Dynamic checks cannot be invoked in the structure, exports, metric or reports section of a contract. As with static checks, run-time checks can be viewed as an operation with a return type and parameter set. A run-time check is guaranteed to be side-effect free.

3.1.3 Metric Evaluators

Metric evaluators are used to analyze and report on metrics gathered during the execution of the IUT. Metric gathering is performed by the framework's profiler in conjunction with metrics gathered within the contract itself. Once metric gathering is complete and the IUT has completed execution, the metric evaluators are executed. Examples of a metric evaluator include: performance, space, and network use analysis. Metric evaluators can only be invoked by placing code in the reports section of the contract. Information regarding the reports contract section will be presented shortly. Metric evaluators cannot be invoked in any other contract section. As metric evaluators do not directly interact with the IUT, they are side-effect free.

3.1.4 Summary

Through the creation and use of static checks, run-time checks, and metric evaluators the ACL can be used to perform checks and tasks which are not directly supported by the language. Such plug-ins are analogous to C++'s standard template library (STL) or plug-ins in Adobe's Photoshop. The remaining sections will illustrate how such plug-ins can be imported and used within an ACL contract.

3.2 Loading Plug-ins

All plug-ins, regardless of type, are packaged within managed dynamic link libraries (DLLs). Each DLL may contain one or more plug-ins. The DLLs must first be registered with the ACL compiler. Registration is accomplished automatically by placing the plug-ins in a predefined location, henceforth known as the plug-in folder.[7] Upon initialization, the ACL compiler will examine the contents of the plug-in folder and all sub-folders for plug-ins. Each time a plug-in is located it is examined via reflection to determine the namespace, signature, and type of the plug-in. Once such examination is complete, the plug-in is registered with the ACL compiler and can be imported into contract files. Only plug-ins that have been successfully registered by the ACL compiler can be imported into contract files. If a plug-in is imported into a contract file which has not been registered, a compile-time error will be issued.

3.3 Namespaces

Each plug-in must be located within a namespace. That is, there is no notion of a root or default plug-in. Each namespace can include any number of plug-ins. Namespaces are composed of one or more namespace elements separated by a dot (.). A namespace element follows the same rules used to define an ACL identifier. Grammar 3-1 illustrates the namespace name declaration and Figure 3-1 illustrates an example of a namespace.

```
namespace-name:
        identifier
        namespace-name . identifier
```

Grammar 3-1 - Namespace name

```
DaveArnold.Examples.GroceryStore
```

Figure 3-1 - Namespace example

Namespaces are used for grouping and ordering of plug-ins. Company names and functionality can be the driver for namespace usage; however the framework does not impose any namespace grouping guidelines.

3.4 Import Declarations

Regardless of type all plug-ins are imported into an ACL contact through the use of an import declaration. An import declaration begins with the use of the ***Import*** keyword. The ***Import*** keyword is followed by a namespace to import and is terminated by a semi-colon (;). Each import must be specified on a separate line of the contract file. Plug-in imports must be specified before any other contract elements. That is, all import declarations must occur at the beginning of an ACL file. Grammar 3-2 specifies the grammar for import declarations.

[7] The physical path for the plug-ins is the "Plugin" folder located relative to the ACL compiler.

```
import-declaration:
      Import namespace-name ;
```

Grammar 3-2 – Import declaration

If an import declaration specifies a namespace which does not exist, a compile-time error will be generated. That is, the specified namespace does not contain any registered plug-ins. Once a namespace has been imported successfully, all plug-ins which reside within that namespace are available for use within any contracts defined in the same contract file.

3.5 Summary

Plug-ins allow for the ACL to be extended with additional checks and functionality that are not defined in this document. Plug-ins are implemented by plug-in developers and not the contract writer. Each plug-in is located in a namespace. Namespaces are imported through the use of an import declaration. Once a plug-in has been imported it can be used within a contract. Import declarations allow for plug-ins to be referenced within a contract. The next section will define how other contracts can be referenced.

4 Using Declarations

A using declaration indicates to the ACL compiler that a previously defined contract will be referenced by a contract following the using declaration. Using declarations allow for grouping and ordering of contracts in several contract files. Using declarations promote contract reuse, as they allow new contracts to use and extend contracts that have already been written. Functionally using declarations behave similar to the import declarations described in the previous section. The only difference is that rather than import registered plug-ins, using declarations import existing contracts.

Using declarations appear after import declarations and before contract definitions. A using declaration begins with the ***Using*** keyword followed by a contract namespace. The using declaration is terminated by a semi-colon (;). Each using declaration must be specified on a separate line of the contract file. Grammar 4-1 specifies the grammar for using declarations.

```
using-declaration:
        Using namespace-name ;
```

Grammar 4-1 – Using declaration

If a contract namespace is specified that does not exist, a compile time error will be generated. It should be noted that Grammar 3-2 and Grammar 4-1 both make reference to the same ***namespace-name*** non-terminal symbol. This is due to the fact that both plug-ins and contracts may exist within a single namespace. However, plug-ins must be referenced by means of import declarations and contracts must be referenced by means of using declarations. That is, if a plug-in and contract residing in the same namespace need to be referenced; both an import and using declaration must be specified. The separation was made to distinguish between a library of plug-ins and a library of reusable contracts. The next section will look at the declaration of an actual contract.

5 Contracts

The central element of the ACL is a contract. A Contract is bound to a type. The type is not defined within the ACL, but rather is part of the IUT. Contracts may contain several elements that describe responsibilities, scenarios, static checks, dynamic checks, metric evaluators, and binding rules. The following subsections will examine each of the elements that are used to construct a contract.

5.1 Namespaces

Each and every contract must reside within a namespace. As previously discussed, using declarations allow existing contracts to be included and referenced within a current contract. In order for a contract to be referenced via a using declaration it must be located within a namespace. Namespaces are denoted by using the ***Namespace*** keyword followed by the namespace name. Immediately following the namespace declaration the open scope operator (***{***) is used to denote the start of the namespace. The namespace terminates with a matching close scope operator (***}***). Grammar 5-1 illustrates the namespace declaration grammar.

```
namespace-declaration:
      Namespace namespace-name { contract-declarations_opt }
```

Grammar 5-1 – Namespace declaration

Namespaces with the same name can be declared more than once. That is, the same namespace name can be used in any number of namespace declarations. The result of such use is a single namespace which contains all the contracts defined in each of the identical namespace declarations. Each contract defined within the same namespace must have a unique name. If more than one contract with the same name is specified, a compile-time error will be generated. As Grammar 5-1 illustrates, it is grammatically possible for a namespace declaration to not contain any contract declarations. While this is syntactically correct, and the ACL compiler will create an empty namespace using the given namespace name, it serves no functional purpose.

There is only a single element that may be declared within a namespace: a contract. The next section will define the declaration of contracts.

5.2 Contract Declarations

A namespace contains zero or more contract declarations. In this section we look at the various aspects which compose a contract declaration. Grammar 5-2 contains the complete grammar for the declaration of a contract.

```
contract-declarations:
      contract-declaration
      contract-declarations contract-declaration

contract-declaration:
      contract-modifiers_opt contract-type identifier
            contract-generics_opt base-contract_opt { contract-elements_opt }

contract-modifiers:
      abstract

contract-type:
      Contract
      MainContract

contract-generics:
      < generic-parameters >

generic-parameters:
      generic-parameter
      generic-parameters , generic-parameter

generic-parameter:
      generic-parameter-type identifier

generic-parameter-type:
      Boolean
      Character
      Field
      Integer
      Method
      Real
      String
      Type

base-contract:
      extends identifier base-contract-generics_opt

base-contract-generics:
      < base-generic-parameters >

base-generic-parameters:
      base-generic-parameter
      base-generic-parameters , base-generic-parameter

base-generic-parameter:
      identifier
      literal
```

Grammar 5-2 – Contract declaration

As shown above a contract declaration consists of several elements. Each of the following subsections will provide detailed information regarding the declaration of a contract.

5.2.1 Modifiers

A contract declaration begins with an optional list of contract modifiers. The ACL only supports a single modifier at the contract level, abstract. A contract that is declared using the ***abstract*** keyword cannot be directly bound to a type within an IUT. That is, the contract must be extended and refined by a sub-contract in order to be used. Abstract contracts are similar to the notion of interfaces found in C# and Java. The most common use for abstract contracts is for defining a common set of responsibilities and scenarios that must be attached to other domain specific contract elements for use.

5.2.2 Contract Types

The ACL defines two types of contracts: regular and main. A regular contract is declared using the ***Contract*** keyword. Main contracts are declared using the ***MainContract*** keyword. Both contracts can contain the same contract elements. The only difference between the two is that a main contract is automatically bound to a type within the IUT at compile-time. Regular contracts are only bound to an IUT counterpart when used. That is, regular contracts are only bound if the contract is used or referenced within the current project.

Due to the fact that main contracts are automatically bound to an IUT type, the abstract modifier cannot be used with contracts declared using the ***MainContract*** keyword.

5.2.3 Contract Names

As shown in Grammar 5-2 immediately following the contract type is an identifier. This identifier represents the name of the contract. Contract names must be unique within the namespace that they are declared in. Two contracts may have the same name if they are declared in different namespaces. Contract names are used when accessing the contract or its members in other locations, binding, and when reporting the results of contract execution. As such, it is good design practice to select meaningful contract names.

5.2.4 Contract Generics

Following the name of the contract, a list of one or more generic parameters may be specified. The use of contract generics when defining a contract is optional. Generics allow a template contract to be specified, and then through the use of generic parameters the contract can be specialized into a concrete contract. For example the contract in Figure 5-1 defines a generic contract that can be applied to a fixed length container. The contract is declared within the ***DaveArnold.Collections*** namespace and is named ***BoundedContainer***. The two generic parameters are used to specify the element type to be stored in the container and the upper bound of the container.

```
Import Core;

Namespace DaveArnold.Collections
{
	Contract BoundedContainer<Type T, Integer MaxSize>
	{
		// Contract Body…
	}
}
```

Figure 5-1 - A generic contract

As defined in Grammar 5-2 and illustrated in Figure 5-1, a contract's generic parameter set immediately follows the name of the contract. A generic parameter set begins with a **<** and consists of one or more generic parameters comma (,) delimited. Finally the generic parameter set ends with a **>**. Each generic contract can contain up to 32 generic parameters.

5.2.4.1 Generic Parameters

Each generic parameter as defined in Grammar 5-2 consists of two elements a type followed by an identifier. The type indicates the generic parameter type, and the identifier is used to reference the parameter within the contract. Any of the ACL built-in types can be used as a generic parameter, with the exception of the ***Timer*** type. The ***Timer*** type cannot be used as a generic parameter because it is a specialized type that does not apply from the generic point of view.[8] IUT types, methods, and fields can also be used as a generic parameter through the use of the ***Type***, ***Method***, and ***Field*** built-in types respectively.

Following the generic parameter type, an identifier is specified to represent the generic parameter within the contract. In effect, the specified identifier becomes a keyword within the body of the contract. The value passed to the generic parameter, will be used to substitute the placeholder represented by the identifier. Each generic parameter list must have unique identifiers. That is, no two generic parameters may have the same name.

5.2.4.2 Generic Implementation

Contract generics are implemented by the ACL compiler. The compiler takes the values passed to the generic parameters and performs substitution. The substitution is performed before any semantic checking is done on the contract. That is, contract generics can be viewed as syntax-sugar to allow templated contracts to be transformed into concrete contracts. The generics promote the reuse of templated contracts, and allow the creation of contract libraries.

5.2.5 Inheritance

The final element in a contract declaration is an extension of a base contract, or inheritance. The extension of an existing contract is optional. Each contract can extend at most one base contract. That is, multiple contract inheritance is not supported. Figure 5-2 provides an example. The example shows the ***BoundedQueue*** contract extending the ***BoundedContainer*** base contract.

[8] That is, we have not found any situations where the ***Timer*** type would be used as a generic parameter.

```
Import Core;

Namespace DaveArnold.Collections
{
      Contract BoundedQueue<Type T, Integer MaxSize>
                                    extends BoundedContainer<T, MaxSize>
      {
            // Contract Body…
      }
}
```

Figure 5-2 - Contract inheritance

Any contract elements that are defined within the base contract are available for use within the contract being declared. As each contract section has different rules regarding inheriting and refining base element, the precise details will be discussed at the contract element level.

Syntactically, inheritance is performed through the use of the ***extends*** keyword. The ***extends*** keyword is then followed by an identifier indicating the name of the contract that will serve as the base for the new contract. The base contract must either reside within the same namespace, or a using declaration must be specified.

5.2.5.1 Parameter Values for Generic Bases

If the base contract contains generic parameters, values for each generic parameter must be specified. There are two methods for the specification of such generic parameters. The first is to literally specify the value. In the case of Figure 5-2 a literal value could have been used to specify the size of the bounded container. The second method for the specification of base generic parameter values is by using a generic parameter from the contract being declared, as is the case in Figure 5-2. Generic parameter values are specified by placing a **<**, followed by the parameter values separated by commas (**,**). The generic parameter values are terminated with a matching **>**.

5.2.6 Summary

Following the contract declaration an open brace (***{***) is used to denote the beginning of the contract's body. A matching close brace (***}***) is used to denote the end of the contract's body and in turn the end of the contract.

5.3 Contract Bodies

A contract body is composed of zero or more contract elements. There is no ordering requirement for the elements found within the contract body. However, some contract elements can only be specified once within the contract. A contract that does not contain any elements is still a valid, but useless, contract. As Grammar 5-3 illustrates there are nine types of contract elements.

```
contract-elements:
      contract-element
      contract-elements contract-element

contract-element:
      variable-declaration
      structure-declaration
      observability-declaration
      invariant-declaration
      responsibility-declaration
      scenario-declaration
      metric-declaration
      reports-declaration
      exports-declaration
```

Grammar 5-3 - Contract body

Before defining each of the nine contract elements in detail, there are a few common expressions and statements that can occur in several of the contract elements. These common items include member access, assignment, and mathematical expressions. Grammar 5-4 provides a formal definition for the common expressions and statements.

```
empty-statement:
      ;

expression-statement:
      statement-expression ;

statement-expression:
      invocation-expression
      assignment

invocation-expression:
      primary-expression ( argument-list_opt )

argument-list:
      argument
      argument-list , argument

argument:
      expression

expression:
      non-assignment-expression
      assignment

non-assignment-expression:
      conditional-or-expression

conditional-or-expression:
      conditional-and-expression
      conditional-or-expression || conditional-and-expression

conditional-and-expression:
      equality-expression
      conditional-and-expression && equality-expression

equality-expression:
      relational-expression
      equality-expression == relational-expression
      equality-expression not == relational-expression
      equality-expression sameas relational-expression

relational-expression:
      additive-expression
      relational-expression <additive-expression
      relational-expression > additive-expression
      relational-expression <= additive-expression
      relational-expression >= additive-expression

additive-expression:
      multiplicative-expression
      additive-expression + multiplicative-expression
      additive-expression – multiplicative-expression

multiplicative-expression:
      unary-expression
      multiplicative-expression * unary-expression
      multiplicative-expression / unary-expression
```

```
unary-expression:
      primary-expression
      + unary-expression
      - unary-expression
      not unary-expression

assignment:
      unary-expression = expression

primary-expression:
      literal
      identifier
      ( expression )
      member-access
      invocation-expression
      context-access
      value-access
      dontcare-expression

member-access:
      primary-expression .  identifier

context-access:
      context

value-access:
      value

dontcare-expression:
      dontcare
```

Grammar 5-4 - General expressions

The ACL handles assignments, invocations, and mathematical expressions in the same fashion as most high-level programming languages. As such, a detailed explanation of the ACL elements defined in Grammar 5-4 is intentionally omitted. Grammar 5-4 obeys the operator precedence rules previously defined. There are a few things to note before continuing. The ***invocation-expression*** is used to either invoke an element of the contract, such as a responsibility, or it is used to invoke a plug-in. The context of the invocation is determined from the context. The ***context-access*** expression is used to reference the current contract instance. The ***context*** keyword functions similarly to the C++ ***this*** keyword. The ***value-access*** expression is used to access the value to be returned by an observability method, responsibility, or metric method. Finally, the ***dontcare-expression*** is used within scenario statements to indicate that the parameters for a particular responsibility to be executed are not important.

Now that the general purpose expressions have been defined, the following sections will define each of the nine contract elements.

5.4 Variables

Contract variables are visible throughout all elements within the contract. They are used to store information during the evaluation of the contract. The previous sections have already discussed variable types, and modifiers. In this section the variable declarations will be defined. Grammar 5-5 illustrates the variable declaration grammar.

```
variable-declaration:
      variable-modifiers_opt list-or-value_opt variable-type identifier ;

variable-modifiers:
      exported
      once
      exported once

list-or-value:
      List
      Value

variable-type:
      identifier
      built-in-type

built-in-type:
      Boolean
      Character
      Field
      Integer
      Method
      Real
      String
      Timer
      Type
```

Grammar 5-5 - Variable declaration

A variable declaration begins with an optional set of modifiers. As previously discussed, there are two modifiers that can be applied to variables: ***exported*** and ***once***. The ***exported*** modifier indicates that the variable is visible from outside the contract. The ***once*** modifier indicates that the variable can only be used as an l-value once. That is, the variable cannot only be assigned to once. List variable types cannot use the ***once*** modifier. Following the variable modifiers is an optional value that indicates the class of variable being declared. As previously discussed, the ACL supports two variable classes: list and value. The list variable class declares an unbounded ordered collection of variables. The value variable class declares a scalar variable. If no variable class is specified, the value class is assumed.

Next the variable type is specified. Variables can either be of one of the built-in types or an element of the IUT. Any of the previously discussed built-in types can be used as a variable type. IUT elements are represented by an identifier. The identifier must then be referenced in an export line found within the ***Exports*** section of the contract. If the identifier used for the variable's type is not found within the ***Exports*** section of the contract, a compile-time error will be issued. Finally, the variable's name is specified via an identifier. The identifier used in the variable name must not already

be in use. That is, each variable name must be unique, and the variable name cannot already have a meaning within the context of the contract. The variable declaration is terminated with a semi-colon (**;**).

5.4.1 Inheritance Rules

Variables declared in a base contract are available for use in any derived contracts, regardless of inheritance depth. The ***exported*** modifier does not have to be specified. The ***exported*** modifier only applies when accessing a variable from outside the contract.[9] Variables defined in sub-contracts cannot use the names of variables defined in base contracts. That is, each variable defined within the inheritance hierarchy must have a unique name.

Variables cannot be overridden, hidden, or refined through inheritance.

[9] External to the inheritance hierarchy.

5.5 Structure

A contract's structure section is used to specify static checks that are to be evaluated as part of the contract. Each contract may at most contain one structure section. Recall that static checks are checks that can be evaluated without executing the IUT. Such checks include checking for inheritance relationships, method existence, and parameter conformance. All static checks must be specified within the structure section of a contract. That is, static checks cannot be placed in any other contract section. The ACL does not define any specific static checks; all static checks are provided as plug-ins.

Syntactically a structure section is denoted by the ***Structure*** keyword followed by an open brace (***{***). The body of the structure section is specified between the open brace and a matching close brace (***}***). The close brace denotes the end of the structure section. Grammar 5-6 defines the declaration of the structure section.

```
structure-declaration:
      Structure structure-body

structure-body:
      { structure-statement-list_opt }

structure-statement-list:
      structure-statement
      structure-statement-list structure-statement
```

Grammar 5-6 - Structure declaration

5.5.1 Inheritance Rules

If a base contract contains a structure section, it will be evaluated before the structure section found in the current contract. That is, the flattened contract will contain a single structure section that is composed of the base's structure section followed with current contract's structure section.

Structure sections cannot be overridden, refined, or hidden through inheritance.

5.5.2 Structure Bodies

As defined in Grammar 5-6, a structure section consists of a body that contains zero or more statements. The statements supported by a structure section are defined in Grammar 5-7.

```
structure-statement:
      variable-declaration
      structure-embedded-statement

structure-embedded-statement:
      structure-body
      empty-statement
      expression-statement
      belief-statement-str
```

Grammar 5-7 - Structure body

There are two key types of statements that can occur within the body of a structure section: variable declarations, and embedded statements. Variable declarations are identical to the ones defined by Grammar 5-5. That is, variables defined within a structure section behave the same as a

contract level variable. The only difference is that variables defined within a structure section only exist within the scope of the structure section. Variables defined within a structure section may have the same name as contract variables. In this case the ***context*** keyword is required in order to access the contract level variables. Variables defined within a structure section must also be declared before they are used within the section. That is, a variable declaration must precede its usage.

The second type of statement that can occur within the body of a structure section is an embedded statement. An embedded statement is any statement that is not a variable declaration. In the case of the structure section, the previously discussed general statements can be used. Practically, this results in the invocation of static checks, and some arithmetic operations. A structure section can also contain a belief statement.

5.5.2.1 Beliefs

Beliefs are used to provide user friendly names and descriptions to a set of checks. Beliefs can be used in several areas of the contract, including the structure section. Beliefs are denoted by the ***Belief*** keyword followed by two parameters: name and description. Belief names are used within the contract evaluation report to reference a given belief. Belief names can be specified using one of two methods. The first method is through the use of an identifier. Each belief specified within the same contract must have a unique identifier. In addition, identifiers that are used for IUT elements and contract variables cannot be used for beliefs. The second method for specifying the name of the belief is through the use of the current contract identifier (**~**) operator. The current contract identifier operator can only be used in beliefs and provides shorthand for referencing the name of the contract. The current contract identifier operator must be followed by either an identifier or an integer value to provide additional naming information. The second parameter for the belief, specified between parentheses, is a string literal which specifies an optional description for the belief. Beliefs are defined in Grammar 5-8.

```
belief-statement-str:
      Belief belief-name belief-description_opt structure-embedded-statement

belief-name:
      identifier
      ~ identifier
      ~ integer-literal

belief-description:
      ( string-literal )
```

Grammar 5-8 - Beliefs

Beliefs are attached to the statement immediately following them. If the next statement is another belief, then the two beliefs are both applied to the next statement, and so on until the first non-belief statement is encountered. That is, any number of beliefs can be grouped together, and applied as a group to the first non-belief element found. If a belief is to be applied to a group of elements, a pair of matching braces (***{*** and ***}***) can be used to denote a scope whose elements will each have the belief applied. The maximum scope depth is 32. Beliefs can also be nested as needed.

Beliefs can either pass or fail. A belief passes if each and every statement to which it is applied succeeds. Otherwise a belief fails. The ACL compiler translates the beliefs into scoped names for use within the contract evaluation report. Figure 5-3 illustrates an example of using a belief within a structure section.

```
Structure
{
      Belief ~1("The customer needs a way to store items
                              in his/her cart")
      {
            HasMemberOfType(tFoodContainer);
      }
}
```

Figure 5-3 - Belief usage

The belief in Figure 5-3 is defined using the current contract identifier operator, and a 1. The belief then contains a description. The belief is attached to the single check specified within the matching braces. If the check passes, the belief will pass. Otherwise the belief will fail.

5.6 Observability Methods

Observability methods are read-only methods that are used to acquire state information about the IUT for evaluation within a contract. A contract can contain any number of observability methods. Observability methods function when the IUT is being executed. That is, they can only be invoked from within an invariant, metric, responsibility, or scenario. Observability methods can be implemented in one of two ways as shown in Grammar 5-9.

```
observability-declaration:
      bound-observability
      defined-observability
```

Grammar 5-9 - Observability declaration

5.6.1 Bound Observability Methods

The first way is by directly binding an observability method to a read-only method within the IUT. The developer of the IUT is responsible for ensuring that the observability methods required by a given contract are implemented within the IUT. Once a method binding for an observability method is completed, static checks are automatically added to the contract to ensure that the corresponding IUT method is actually read-only. That is, the binding operation triggers the addition of static checks to ensure that the target method is in-fact side effect free. As the behaviour of the observability method is provided by the IUT, the contract only requires the signature specification of the observability method. Grammar 5-10 defines bound observability declarations.

```
bound-observability:
      observability-modifers_opt Observability return-type identifier
            parameter-list ;

observability-modifiers:
      refine
      abstract

return-type:
      list-or-value_opt return-type-names-no-void

return-type-names-no-void:
      identifier
      built-in-type

parameter-list:
      ( parameters_opt )
      ( Void )

parameters:
      parameter
      parameters , parameter

parameter:
      variable-type identifier
```

Grammar 5-10 - Bound observability declaration

5.6.2 Defined Observability Methods

The second way to implement an observability method is by providing a body. If an observability method contains a body, it is said to be a defined observability method. Defined observability methods invoke other observability methods to determine their value, rather than being bound to an IUT method. That is, defined observability methods are not bound to the IUT. Defined observability methods are defined by Grammar 5-11.

```
defined-observability:
      observability-modifers_opt Observability return-type identifier
            parameter-list observability-body

observability-body:
      { observability-statement-list }

observability-statement-list:
      observability-statement
      observability-statement-list observability-statement

observability-statement:
      variable-declaration
      observability-embedded-statement

observability-embedded-statement:
      observability-body
      empty-statement
      expression-statement
      value-statement
```

Grammar 5-11 - Defined observability

5.6.3 Observability Signatures

Regardless of the observability type, the same observability signature is used. An observability signature begins with an optional modifier: ***refine***. Details regarding the ***refine*** modifier will be presented shortly. Following the modifier, the ***Observability*** keyword is used to indicate that the signature is defining an observability. Next, the return type of the observability is specified. As the purpose of an observability method is to return a value to the caller, a return type must be specified, and the ***Void*** type cannot be used. The return type can either be one of the previously discussed built-in types or an identifier which is bound to an IUT type. After the return type is specified an identifier is used to denote the name of the observability method. Each observability must have a unique name. Finally, an optional parameter set is specified between parenthesis. Each parameter is separated by a comma (**,**). Each parameter is defined by using a built-in type or an identifier that is bound to an IUT type, followed by an identifier. Figure 5-4 provides examples of observability methods.

```
Observability Boolean   IsFull();
Observability Boolean   IsEmpty();
Observability T         ItemAt(Integer index);
Observability Boolean   HasItem(T item);
Observability Integer   Size();
Observability Integer   BackLocation();
Observability Integer   FrontLocation();
Observability T         Back()
{
      ItemAt(BackLocation());
}
Observability T         Front()
{
      ItemAt(FrontLocation());
}
```

Figure 5-4 - Observability examples

5.6.4 Inheritance Rules

Observability methods defined within a base contract are visible by any contract derived from the base. Observability methods that are marked with the ***abstract*** modifier must be implemented in a derived contract. If the ***abstract*** modifier is used the contract where the abstract observability method is defined, must also be marked abstract. That is, if a base observability method is marked abstract, it must be refined as a defined observability in a derived contract. All non-abstract bound observability methods defined within a base contract will be bound to corresponding IUT methods during the binding of the derived contract. If an observability method in a derived contract wishes to refine an observability method defined in a base contract, the ***refine*** modifier can be used. The ***refine*** modifier is used to explicitly indicate that the following observability method is replacing the base observability method with a matching observability signature. Refined observability methods must be a defined observability. That is, an observability cannot be refined into a binding. If the ***refine*** modifier is used in a situation where there is no matching base observability a compile-time error will be issued.

Observability methods cannot be hidden through inheritance.

5.6.5 Observability Bodies

Defined observability methods contain a body. The purpose of the body is to calculate and return the requested observability value. Such calculation can be accomplished by performing mathematical calculations and invoking other observability methods as defined in Grammar 5-4. Observability methods are prevented from invoking metrics, responsibilities, and scenarios. In addition, any behaviour defined in the body must be side effect free with respect to the IUT. That is, the observability method may modify the state of the contract, but not the state of the IUT.

5.6.5.1 Value Statements

As defined by the observability signature, the body must return a value. If the body is implemented as a single statement or expression, it is the value of executing the single line that is used as the return value. Otherwise a value statement must be specified. A value statement is similar to an assignment, where the ***value*** keyword is assigned a value. It is this value that is returned to the caller.

Grammar 5-12 defines the value statement. An observability body that is defined as more than a single statement and that does not contain a value statement will generate a compile-time error.

```
value-statement:
      value = expression ;
```

Grammar 5-12 - Value statement

5.7 Invariants

Invariants represent a set of one or more checks that are applied at the beginning and end of each responsibility defined within the contract. There are two exceptions to this rule. The first exception is that invariants are not checked at the beginning of the ***new()*** responsibility. The second exception is that invariants are not checked at the end of the ***finalize()*** responsibility. Details regarding both ***new()*** and ***finalize()*** responsibilities will be presented in the next section. A contract can contain any number of invariants. When more than one invariant is present within a contract, the ACL compiler will merge the invariants into a single invariant, with multiple named sections. As illustrated in Grammar 5-13, each invariant is denoted by a name. The name of an invariant must be unique. That is, two invariants may not have the same name. It is this name that will be used in the contract evaluation report to identify which invariant failed. Beliefs can also be used within an invariant body.

```
Invariant-declaration:
      invariant-modifers_opt Invariant invariant-body

invariant-modifiers:
      refine
      abstract
```

Grammar 5-13 - Invariant declaration

5.7.1 Inheritance Rules

Invariants that are defined within a base contract are automatically applied to the responsibilities in a derived contract. Invariants can be defined abstract only if the containing contract is marked abstract. Abstract invariants must be refined within a derived contract. If any abstract invariants are not refined a compile-time error will be generated. Any base invariant, abstract or not, can be refined. Refinement is accomplished by using the ***refine*** modifier along with the same invariant name. A refined invariant can add additional checks to the base invariant; however the functionality of the base invariant cannot be changed. That is, an invariant may be strengthened but not weakened.

Invariants cannot be overridden, or hidden through inheritance.

5.7.2 Invariant Bodies

The purpose of an invariant's body is to define the checks that are to be applied to the responsibilities before and after execution. Invariants can define variables to aid in the checks. The variable declarations are identical to the contract variables previously discussed, except the ***exported*** modifier cannot be used, and the variable scope is only within the invariant. In addition, as defined in Grammar 5-14 beliefs can be used to provide user friendly names and descriptions to a set of checks. The beliefs used within invariant bodies are identical to the beliefs used in the previously discussed structure section. The only difference is that the body of an invariant belief contains invariant checks, and not structural ones. Grammar 5-15 illustrates the grammar changes.

```
invariant-body:
      { invariant-statement-list_opt }

invariant-statement-list:
      invariant-statement
      invariant-statement-list invariant-statement

invariant-statement:
      variable-declaration
      invariant-embedded-statement

invariant-embedded-statement:
      invariant-body
      empty-statement
      expression-statement
      belief-statement-inv
      check-statement
```

Grammar 5-14 - Invariant body

```
belief-statement-inv:
      Belief belief-name belief-description_opt invariant-embedded-statement
```

Grammar 5-15 - Invariant beliefs

5.7.2.1 Check Statements

The actual invariant checks are performed by using a check statement. Check statements define a condition that is to be tested by the invariant. If the condition evaluates to true, then the check succeeds, otherwise the check fails. An invariant may contain any number of check statements. The check statement is defined in Grammar 5-16. The statement begins with the ***Check*** keyword followed by a condition body enclosed with parenthesis. As defined in Grammar 5-16 the condition body is represented by an expression. The expression must evaluate to a Boolean value. In addition, the ***pass*** and ***fail*** keywords can be used to denote a check that automatically passes or fails.

```
check-statement:
      Check ( check-condition-body ) ;

check-condition-body:
      expression-statement
      pass
      fail
```

Grammar 5-16 - Check statement

5.8 Responsibilities

A contract may contain any number of responsibilities. A responsibility can be viewed as a functional requirement, or a functional unit. Responsibilities can be used in one of two ways. The first and most common way is to bind the responsibility to a method found within the IUT. The second way is to define the responsibility through the use of a grammar which in-turn uses other responsibilities for the definition. Regardless of how the responsibility is used, responsibilities are denoted by a method like signature: optional modifiers, a return type, name, and optional parameters. Responsibility declaration is defined by Grammar 5-17.

```
responsibility-declaration:
      responsibility-modifers_opt Responsibility return-type_opt
            responsibility-name parameter-list responsibility-body
      abstract Responsibility return-type_opt responsibility-name
            parameter-list ;

responsibility-modifiers:
      refine

return-type:
      list-or-value_opt return-type-names

return-type-names:
      identifier
      built-in-type
      Void

responsibility-name:
      identifier
      new
      finalize

parameter-list:
      ( parameters_opt )
      ( Void )

parameters:
      parameter
      parameters , parameter

parameter:
      variable-type identifier
```

Grammar 5-17 - Responsibility declaration

There are two types of responsibility declarations: regular and abstract. Regular responsibility declarations contain a responsibility body, where abstract responsibilities do not. Details between the two types will be discussed in the next section. After any optional modifiers the ***Responsibility*** keyword is used to denote a responsibility definition. Next an optional return type is specified. The return type indicates the type of value that is returned by the responsibility. If no return type is specified the ***Void*** type is assumed. That is, no value is returned by default. The actual value to return is the value returned by the IUT method bound to the responsibility. Each responsibility is identified by a unique name. The responsibility name does not have to match any names found within the IUT. However, no

two contract elements may have the same name. That is, a responsibility cannot have the same name as another invariant, observability method, responsibility, scenario, or metric.

5.8.1 Special Responsibilities

There are two keywords that can be used in place of an identifier for the responsibility name: ***new*** and ***finalize***. Each keyword denotes a special type of responsibility. These special responsibilities can only be used by binding to an actual IUT method, and cannot be defined through the use of other responsibilities. The following subsections will look at each of the special responsibilities.

5.8.1.1 The New Responsibility

The new responsibility is denoted by using the ***new*** keyword to name the responsibility. Each contract may have at most one new responsibility. The new responsibility does not require an explicit binding, but rather the responsibility is automatically evaluated following the creation of a new instance that is of the type bound to the containing contract. That is, the new responsibility is bound to each and every constructor found within the IUT type bound to the containing contract. Otherwise, the new responsibility behaves exactly as any other responsibility with one exception: the new responsibility cannot define any preconditions within its body.

5.8.1.2 The Finalize Responsibility

Analogous to the new responsibility is the finalize responsibility. The finalize responsibility is denoted by the ***finalize*** keyword. Each contract may have at most one finalize responsibility. Like the new responsibility, the finalize responsibility does not require an explicit binding, but rather the responsibility is automatically evaluated before the destruction of an instance which is of the type bound to the containing contract. That is, the finalize responsibility is bound to each and every destructor found within the IUT type bound to the containing contract. Otherwise, the finalize responsibility behaves exactly as any other responsibility with one exception: the finalize responsibility cannot define any post-conditions within its body.

Finally, as with observability methods, responsibilities contain an optional parameter list enclosed with parenthesis. If the responsibility accepts no parameters an empty parenthesis must be specified. If the responsibility is defined using the ***abstract*** modifier the responsibility declaration is terminated with a semi-colon (***;***), otherwise a responsibility body is specified. Figure 5-5 provides an example of two responsibilities.

```
Responsibility Add(T item)
{
      Pre(IsFull() == false);
      Pre(item not= null);
      Pre(HasItem(item) == false);
      context.size = context.size + 1;
      item_timer.Start(item);

      Post(HasItem(item) == true);
}

Responsibility T Remove()
{
      Pre(IsEmpty() == false);

      item_timer.Stop(value);
      context.size = context.size - 1;
      Post(value not= null);
      Post(HasItem(value) == false);
}
```

Figure 5-5 – Responsibilities

The first responsibility is named ***Add*** and as a single parameter of type ***T***. The body of the responsibility ensures that the collection is not full, and that item to be added is valid, not null, and does not already exist within the collection. The responsibility then increments a contract variable representing the size of the collection. Next, a timer is started to track how long the element is stored within the collection. Finally a post-condition is specified to ensure that the item was actually added to the collection.

The second responsibility named ***Remove*** has a return type of ***T*** and accepts zero parameters. The responsibility first checks to see if there is an item to remove from the collection. Next, the ***value*** keyword is used to get the value returned by the bound IUT method. The value is used to stop the timer tracking how long the item was stored within the collection. The contract variable representing the size of the collection is then decremented. Finally, post-conditions check to see if an actual item was removed from the collection and that the item is no longer stored within the collection.

5.8.2 Inheritance Rules

Abstract responsibilities are defined using the ***abstract*** modifier. The ***abstract*** modifier can only be used if the containing contract is also marked abstract. Any responsibilities that are marked abstract must be refined in a derived contract. Any abstract responsibilities that are not refined will result in a compile-time error. Any responsibilities that are defined within a base contract are available for use within all derived contracts. A base responsibility may be refined by using the ***refine*** modifier along with a matching responsibility signature. The refined responsibility may add additional checks to the base responsibility, but not remove any. That is, responsibilities follow design-by-contract rules, in that any preconditions may be weakened but not strengthened. In addition post-conditions can be strengthened but not weakened.

Responsibilities cannot be overridden, or hidden through inheritance.

5.8.3 Responsibility Bodies

The purpose of a responsibility's body is to define the checks that are applied before and after the IUT method that implements the responsibility executes. Additional calculations and updates to the contract variables can also be performed. Responsibilities can also define their own variables to be used within a check. These variable declarations are exactly the same as the contract variables except the variable scope is limited to the body of the responsibility. The ***exported*** modifier can be used to provide access to a variable from outside the responsibility. Grammar 5-18 defines the grammar for a responsibility body. As previously stated there are two types of responsibilities, one to which an IUT method is bound, and one that is defined via a grammar of other responsibilities. The type of responsibility is determined by the presence or absence of a scenario statement. Scenario statements are just one of the statements that can occur within a responsibility body. The following subsections, will define each of the statements which can compose a responsibility.

```
responsibility-body:
      { responsibility-statement-list_opt }

responsibility-statement-list:
      responsibility-statement
      responsibility-statement-list responsibility-statement

responsibility-statement:
      variable-declaration
      responsibility-embedded-statement

responsibility-embedded-statement:
      responsibility-body
      empty-statement
      expression-statement
      belief-statement-resp
      pre-statement
      pre-set-statement
      post-statement
      scenario-statement
```

Grammar 5-18 - Responsibility body

5.8.3.1 The Belief Statement

The first specific statement type defined in Grammar 5-18 is the belief statement. Beliefs within responsibilities, behave the same way as the beliefs found in the other contract sections. The only difference is that the grammar for beliefs within responsibility bodies is different to allow responsibility statements to be placed within the body of the belief. Grammar 5-19 defines beliefs for use within responsibilities.

```
belief-statement-resp:
      Belief belief-name belief-description_opt
            responsibility-embedded-statement
```

Grammar 5-19 - Responsibility beliefs

5.8.3.2 The Pre Statement

The Pre statement is used to represent a precondition. Preconditions are used to specify a check that is evaluated before the responsibility is executed. Depending on the presence of a scenario

statement, this evaluation will either take place before the bound IUT method is executed or before the responsibility grammar is executed. Such checks can be used to test the parameter values passed to the responsibility, or the state of both the contract and the IUT before executing the responsibility. Grammar 5-20 defines a precondition. The statement begins with the ***Pre*** keyword followed by a condition body enclosed with parenthesis. As with the previously defined ***Check*** keyword, preconditions are specified using an expression which results in a Boolean value. Finally the precondition is terminated with a semi-colon (**;**).

```
pre-statement:
      Pre ( pre-condition-body ) ;

pre-condition-body:
      expression-statement
      pass
      fail
```

Grammar 5-20 - Pre statement

5.8.3.3 The PreSet Statement

The PreSet statement is used to attain a value before the responsibility is executed. This value can then be used within the body of the responsibility or within a Post statement. The PreSet statement is defined in Grammar 5-21. The statement begins with an identifier that must have already been defined as either a contract variable or a variable within the responsibility. This variable will be used to store the value returned by the PreSet expression. After the identifier and the assignment (**=**) operator, the ***PreSet*** keyword is specified to indicate that we wish to assigned the variable the value returned by the specified condition. The condition itself is specified using an expression. This condition is evaluated before the responsibility is executed.

```
pre-set-statement:
      identifier = PreSet ( pre-set-condition-body ) ;

pre-set-condition-body:
      expression-statement
```

Grammar 5-21 - PreSet statement

5.8.3.4 The Post Statement

The post statement is used to represent a post-condition. Post-conditions are used to specify a check that is evaluated after the responsibility has executed. Depending on the presence of a scenario statement, this evaluation will either take place after the bound IUT method has been executed or after the responsibility grammar has been executed. Such checks can be used to test that the responsibility executed correctly and that any return value is correct. The post statement is defined by Grammar 5-22, and beings with the ***Post*** keyword followed by a condition body enclosed with parenthesis. As with the ***Check***, ***Pre***, and ***PreSet*** statements post-conditions are specified using a Boolean expression. The post statement is terminated with a semi-colon (**;**).

```
post-statement:
      Post ( post-condition-body ) ;

post-condition-body:
      expression-statement
      pass
      fail
```

Grammar 5-22 - Post statement

5.8.3.5 The Scenario Statement

The scenario statement indicates that the responsibility is not bound to an actual IUT method, but rather is defined by a scenario. That is, the responsibility is defined by its body rather than an IUT method. This scenario consists of a grammar of responsibilities. The scenario is satisfied when the execution of the IUT matches the specified grammar of responsibilities. In the case of a responsibility, the responsibility is fulfilled if the specified grammar of responsibilities is satisfied. If the grammar of responsibilities is not satisfied then the responsibility will fail. Each responsibility may have at most one scenario statement. If more than one scenario statement occurs a compile-time error will be generated. If no scenario statement is specified, then the responsibility will be bound to an IUT method. Grammar 5-23 defines the scenario statement. The following subsections will define the scenario statement in more detail.

```
scenario-statement:
      scenario-element ;

scenario-element:
      temporal-element
      scenario-element , temporal-element

temporal-element:
      logical-or-element
      atomic { scenario-element }
      parallel { scenario-element }

logical-or-element:
      logical-and-element
      logical-or-element | logical-and-element

logical-and-element:
      one-or-more-element
      logical-and-element & one-or-more-element

one-or-more-element:
      zero-or-more-element
      one-or-more-element +

zero-or-more-element:
      fixed-scenario-element
      zero-or-more-element *

fixed-scenario-element:
      primary-scenario-element
      fixed-scenario-element [ integer-literal ]

primary-scenario-element:
      ( scenario-element )
      scenario-element-embedded-statements

scenario-element-embedded-statements
      scenario-element-embedded-statement
      scenario-element-embedded-statements
            scenario-element-embedded-statement

scenario-element-embedded-statement:
      variable-declaration
      empty-statement
      expression-statement
      belief-statement-resp
      check-statement
      new-instance-statement
```

Grammar 5-23 - Scenario statement

5.8.3.5.1 Scenario Elements

A scenario statement begins with a scenario element followed by a semi-colon (**;**). The semi-colon is used to denote the end of the scenario statement. That is, the scenario succeeds when the scenario's execution encounters the semi-colon. A scenario must contain one or more scenario elements. Each scenario element is separated by the follows (**,**) operator. The follows operator

indicates that the scenario element that is located before the follows operator must occur before the scenario element that occurs after the follows operator. The example shown in Figure 5-6 shows a simple scenario that is composed of two scenario elements: ***customer.RemoveItems()***, and ***customer.Pay()***. The use of the follows operator states that the ***customer.RemoveItems()*** responsibility must occur before the ***customer.Pay()*** responsibility.

```
Responsibility ProcessCustomer(tCustomer customer)
{
      customer.RemoveItems(),
      customer.Pay();
}
```

Figure 5-6 - Follows operator example

5.8.3.5.2 Temporal Elements

Within each scenario element, temporal elements can be used to define how the scenario element should execute within the individual scenario element. There are two temporal elements that are supported by the ACL: ***atomic*** and ***parallel***. The following subsections will describe each of the two elements.

5.8.3.5.2.1 Atomic

The atomic element is used to indicate that all responsibilities and other contract statements are to be executed and evaluated as one functional unit. That is, the contents of the atomic element are to be viewed as an atomic unit from the scenario's point of view. As defined in Grammar 5-23, the atomic element is denoted by the ***atomic*** keyword. The scenario elements that compose the atomic element are then specified between a set of matching braces (***{and }***). If any part of the atomic unit fails, the entire unit is said to fail. Atomic elements cannot be nested nor can they contain parallel elements. Figure 5-7 extends the example shown in Figure 5-6 to add an atomic element. The atomic element occurs after the ***customer.Pay()*** responsibility, and consists of updating the ***processed_customers*** contract variable, and executing the ***customer.LeaveStore()*** as one atomic unit.

```
Responsibility ProcessCustomer(tCustomer customer)
{
      customer.RemoveItems(),
      customer.Pay(),
      atomic
      {
            processed_customers = processed_customers + 1;
            customer.LeaveStore(dontcare)
      };
}
```

Figure 5-7 - Atomic example

5.8.3.5.2.2 Parallel

The parallel element is used to specify that all responsibilities and other contract statements can be viewed as a single responsibility, enclosed with the one or more operator. That is, multiple instance of the same scenario element may be executing in parallel at any given time. A parallel section can also be viewed as a sub-scenario within the main scenario, where there can be any number of sub-scenarios active at any one time. One thing to note is that, all sub-scenarios must complete before a scenario

element located outside of the parallel element is executed. As defined in Grammar 5-23, the parallel element is denoted by the ***parallel*** keyword. The scenario elements that compose the parallel element are specified between a set of matching braces (***{and }***). Figure 5-8 continues our running example, by adding a parallel element to the beginning of the scenario statement. The parallel element contains three responsibilities: ***customer.SelectItem()***, ***customer.ReadItem()***, and ***customer.AddItem()***. The parallel element indicates that it is possible for the customer to be reading one item, while selecting another item. That is, several sub-scenarios can be running within the parallel element at any given time. However, all of the sub-scenarios must be finished before the ***customer.RemoveItems()*** responsibility is executed. Otherwise the scenario will fail.

```
Responsibility ProcessCustomer(tCustomer customer)
{
      parallel
      {
            once Value tItem item;
            item = customer.SelectItem(),
            customer.ReadItem(item),
            customer.AddItem(item)
      },
      customer.RemoveItems(),
      customer.Pay(),
      atomic
      {
            processed_customers = processed_customers + 1;
            customer.LeaveStore(dontcare)
      };
}
```

Figure 5-8 - Parallel example

5.8.3.5.3 Logical Elements

Scenario elements can be combined using one of two logical elements: or (**/**) and and (**&**). The following subsections will look at each of the scenario logical elements.

5.8.3.5.3.1 Or

The or (**/**) operator is used to specify that multiple scenario elements can occur during the current scenario element. Only one of the elements must occur for the scenario to continue. If both of the elements occur, the scenario will fail. Figure 5-9 continues our example, by adding the or operator. The example grammar indicates that once the ***customer.ReadItem()*** responsibility has executed, either the ***customer.AddItem()*** or the ***customer.DiscardItem()*** responsibility follows. If both the ***customer.AddItem()*** and the ***customer.DiscardItem()*** responsibilities execute the grammar will fail.

```
Responsibility ProcessCustomer(tCustomer customer)
{
      parallel
      {
            once Value tItem item;
            item = customer.SelectItem(),
            customer.ReadItem(item),
            customer.AddItem(item) | customer.DiscardItem(item)
      },
      customer.RemoveItems(),
      customer.Pay(),
      atomic
      {
            processed_customers = processed_customers + 1;
            customer.LeaveStore(dontcare)
      };
}
```

Figure 5-9 - Logical or example

5.8.3.5.3.2 And

The and (***&***) operator is used to specify that multiple scenario elements can occur during the current scenario element. All of the scenario elements must occur before the grammar is satisfied. That is, unlike the or operator where only a single element has to execute, the and operator requires that all of the scenario elements execute. The order of execution does not matter. In Figure 5-9, if the or operator was replaced with the and operator, the ***customer.AddItem()*** and the ***customer.DiscardItem()*** responsibilities would both have to execute, in any order, before the ***customer.RemoveItems()*** responsibility could execute. An execution tree other than the once specified by the grammar, would result in the scenario failing.

5.8.3.5.4 Repetition Operators

When specifying scenario elements, it is common to have a scenario element that will be repeated a number of times. Sometimes the number of times to repeat is known, others it is not. The ACL supports three repetition operators that can be used with scenario elements. All three repetition operators are applied following the scenario element to which they are being applied. The following subsections will define each of the three repetition operators.

5.8.3.5.4.1 One or More

The one or more (***+***) operator can be attached to a scenario element to indicate that the given scenario element must occur one or more times. That is, the scenario element must be executed at least once. Figure 5-10 illustrates an example of the one or more operator. The grammar specified by the example, indicates that the customer may select one or more items via the parallel element.

```
Responsibility ProcessCustomer(tCustomer customer)
{
      parallel
      {
            once Value tItem item;
            item = customer.SelectItem(),
            customer.ReadItem(item),
            customer.AddItem(item) | customer.DiscardItem(item)
      }+,
      customer.RemoveItems()*,
      customer.Pay()[1],
      atomic
      {
            processed_customers = processed_customers + 1;
            customer.LeaveStore(dontcare)
      };
}
```

Figure 5-10 - Repetition example

5.8.3.5.4.2 Zero or More

The zero or more (***) operator can be attached to a scenario element to indicate that the given scenario element can occur any number of times, including zero times. The example shown in Figure 5-10, uses the zero or more operator to indicate that the ***customer.RemoveItems()*** responsibility may execute any number of times. It is possible that the customer will not have any items to remove, as the grammar shown in the example does not guarantee that the ***customer.AddItem()*** responsibility will be executed.

5.8.3.5.4.3 Fixed

The fixed (***[n]***) operator can be attached to a scenario element to indicate that the given scenario element must occur a specific number of times. The number of times to repeat is specified by an integer literal value. Variables cannot be used for the ***n*** value because the number of times to repeat must be known before the scenario is executed. The value for ***n*** must be greater than zero. In Figure 5-10 the fixed operator is used to indicate that the ***customer.Pay()*** responsibility is executed exactly one time. Of course this is the default value, as all scenario elements execute once before the follows (**,**) operator is processed.

5.8.3.5.5 Element Ordering

Grammar 5-23 defines scenario operator precedence; this is the same precedence as shown in Table 2-5. Parentheses can be used to override the scenario operator precedence, and to group scenario elements.

5.8.3.5.6 Scenario Element Composition

Each scenario element can be composed of several statements as defined in Grammar 5-23. Each scenario element must at minimum contain a responsibility invocation. That is, an actual responsibility that defines the scenario element must be specified. In addition to the responsibility, the scenario element may declare variables, perform mathematical operations on contract and local variables, use beliefs, execute checks, and wait for new instances to occur. Only the last statement type is new, all of the others have been previously discussed and will not be repeated here with the

exception of a few notes. Any variables defined within a scenario element only exist within the lifetime of the scenario element. Scenario element variables cannot use the ***exported*** modifier. Checks are evaluated only after the responsibility has been evaluated. The new instance statement will be defined in the following section.

5.8.3.5.7 The New Instance Statement

The new instance statement is used within a scenario element, to indicate that instead of having a responsibility whose execution defines the scenario element, the creation of a new instance defines the scenario element. The creation of the new instance is performed by the execution of the IUT. Grammar 5-24 defines the new instance statement. The statement begins with an identifier that must map to a previously declared variable. The variable can either be declared at the contract or at the local level. Following the assignment (**=**) operator the ***newInstance*** keyword is used to define the new instance statement. The second identifier must be mapped to an IUT type in the contract's exports section. The IUT type must also be the same type as the first identifier. The use of different IUT types will result in a compile-time error. In addition, use of the built-in types with the ***newInstance*** keyword is forbidden. Finally, the new instance statement is terminated with a semi-colon (**;**). The scenario element is considered to be executed following the creation of a new instance of the specified type, and the assignment of that instance to the variable specified by the first identifier.

```
new-instance-statement:
      identifier = newInstance identifier ;
```

Grammar 5-24 - New Instance

5.8.4 Summary

A responsibility can be viewed as a functional requirement, or a functional unit. Responsibilities can either be implemented by binding the responsibility to a method within the IUT or by including a scenario statement within the body of the responsibility. A scenario statement specifies a grammar that is composed of scenario elements. The grammar is composed of an ordering of responsibilities. If during the execution of the IUT, the responsibilities are executed in such an order as to satisfy the grammar, then the responsibility is said to have executed. In the case where a responsibility contains precondition or post-condition that fails, or a scenario grammar that cannot be satisfied the responsibility is said to have failed, and a corresponding message will be displayed in the contract evaluation report. As we have seen responsibilities can be defined in terms of other responsibilities creating a scenario. We will now look at a contract element that defines a scenario from begin to end, rather than just at the responsibility level.

5.9 Scenarios

Unlike responsibilities that represent a functional unit or requirement, a scenario represents a sequence of events or responsibilities that when executed in a specified order, accomplish a task. A scenario section of a contract, allows for the specification of such a sequence. The sequence is specified by using the scenario statement defined in Grammar 5-23. In addition to the scenario statement, a scenario also must have a triggering event and a termination event. These two events define the beginning and end of the scenario respectively. A contract may contain any number of scenario definitions, and each scenario may have any number of active instances. That is, a scenario defines a set of events that represent a task, during execution of the IUT any number of concurrent tasks may running. As with responsibilities, scenarios are composed of a scenario definition and a scenario body. The following subsections will define both elements of a scenario.

5.9.1 Scenario Declarations

Scenarios are declared in a similar fashion to responsibilities, except that scenarios do not have a return type and do not accept parameters. As scenarios are not invoked like responsibilities they cannot accept parameters or return values, rather a scenario is automatically invoked when its triggering condition occurs during the execution of the IUT. Grammar 5-25 defines the scenario declaration.

```
scenario-declaration:
      scenario-modifers_opt Scenario identifier ( ) scenario-body
      abstract Scenario identifier ( ) ;

scenario-modifiers:
      refine
```

Grammar 5-25 - Scenario declaration

Scenarios support two types of declarations: regular and abstract. Regular scenario declarations contain a scenario body, abstract scenarios do not. Details between the two types will be discussed in the next section. After any optional modifiers the ***Scenario*** keyword is used to denote a scenario definition. Next the name of the scenario is denoted by the use of an identifier. Each scenario defined within a given contract must have a unique name. In addition, a scenario may not have the same name as an invariant, responsibility, observability method, or metric. Duplicate names will result in a compile-time error.

5.9.2 Inheritance Rules

Abstract scenarios are defined using the ***abstract*** modifier. An abstract scenario can only be defined within a contract that is also marked abstract. All scenarios that are marked abstract must be refined in a derived contract. Any non-refined scenario will generate a compile-time error. All scenarios that are defined within a base contract also become part of the derived contract. Such base scenarios can be refined by using the ***refine*** modifier along with a matching scenario name. The refined scenario replaces the base scenario. That is, the base scenario is hidden through refinement. The rationale here is that it is impossible to determine if a given scenario strengthens or weakens another. The compiler will issue a warning when a non-abstract scenario is refined. The purpose of the warning is to alert the user that the base scenario is being overridden.

5.9.3 Scenario Bodies

As with responsibility bodies, a scenario body can define checks that are applied before and after the scenario has been executed the IUT. Additional calculations and updates to contract variables can also be performed. Scenarios can also define their own variables for use within the scenario. These variables are known as scenario variables. Scenario variables are declared and used the exact same way as contract variables, except that the variable scope is limited to the scenario. The ***exported*** modifier can be used to provide access to a variable from outside the scenario. It should be noted that because a scenario can be executed several, possibly concurrent, times that each scenario instance maintains a separate set of scenario variables. When the ***exported*** modifier is used, the scenario variable that the modifier is attached must contain the same value across all scenario instances when being accessed outside the scenario. If such a state does not occur, a run-time error will be generated. That is, an exported scenario variable must maintain a consistent state across all active instances of that scenario.

Scenario bodies must contain at least two statements: trigger and terminate. The trigger statement is used to denote the responsibility that begins the scenario. Any scenario that does not specify a trigger statement can never be executed, and as such will result in a compile-time error being issued. The terminate statement performs the opposite function and is used to denote the responsibility that ends the scenario. Any scenario that does not specify a terminate statement can never terminate, and will result in a compile-time error being issued. Both trigger and terminate statements can occur within atomic blocks as illustrated in Figure 5-11.

```
refine Scenario AddAndRemove()
{
      once Value T x;
      once Value Integer index;

      atomic
      {
            Trigger(Add(x)),
            context.inCount = context.inCount + 1;
            index = context.inCount;
      },
      atomic
      {
            Terminate(x == Remove()),
            context.outCount = context.outCount + 1;
      };
      Post(index == context.outCount);
}
```

Figure 5-11 - Scenario example

The scenario definition in Figure 5-11 provides an example of the ***refine*** modifier. The scenario named ***AddAndRemove*** is either implementing an abstract scenario or replacing an existing base scenario. The scenario's body begins with the declaration of two scenario variables. The first scenario variable is of type ***T*** that will be defined in the contract's exports section. The second scenario variable is of the built-in ***Integer*** type. Both scenario variables are scalar in nature and can only be assigned to once, due to the use of the ***Value*** and ***once*** keywords respectively. The scenario is triggered by the invocation of the ***Add(x)*** responsibility. Once the triggering event occurs, the atomic statement ensures

that the contract and scenario variables are also updated. Upon invocation, the scenario variable, ***x***, is assigned the value of the single parameter used in the invocation of the ***Add(x)*** responsibility. The scenario terminates when that same element is removed via the ***Remove()*** responsibility. Again, an atomic statement is used to ensure that the contract variable, ***outCount***, is also updated. Finally, the scenario contains post-condition to ensure that the scenario executed correctly.

Taking a step back from the scenario shown in Figure 5-11, a separate scenario instance is created each time an element is passed to the ***Add(x)*** responsibility, the scenario instance then terminates when the same element is returned as a result to the invocation of the ***Remove()*** responsibility. If the element is never returned, then the scenario will never terminate, and any scenarios that are still running when the IUT terminates will fail.

A scenario is said to succeed if after the triggering responsibility is executed, the execution tree produced by executing the IUT type bound to the contract matches the grammar defined in the scenario body until the terminating responsibility is executed. Responsibilities that are executed outside of the scenario will result in the failure of the scenario, as will having a scenario whose termination condition does not execute. In addition, if the scenario body specifies a ***Check*** statement that fails, the containing scenario will also fail. Scenarios which are specified within a contract that do not execute during the executing of the IUT are simply ignored by the framework, and as such will not succeed or fail. The formal definition of a scenario body is shown in Grammar 5-26. Details of each statement that can be used to compose a scenario will be presented in the following subsections.

```
scenario-body:
      { scenario-body-statement-list }

scenario-body-statement-list:
      scenario-body-statement
      scenario-body-statement-list scenario-body-statement

scenario-body-statement:
      variable-declaration
      scenario-embedded-statement

scenario-embedded-statement:
      scenario-body
      empty-statement
      expression-statement
      belief-statement-scenario
      pre-statement
      pre-set-statement
      post-statement
      check-statement
      scenario-statement
```

Grammar 5-26 - Scenario body

5.9.3.1 The Belief Statement

The first specific statement type defined in Grammar 5-26 is the belief statement. Scenario beliefs are exactly the same as beliefs found within the other contract sections. The only difference is that the grammar for beliefs within scenarios is different to allow scenario statements to be placed within the body of a belief. Grammar 5-27 defines beliefs for use within scenarios.

```
belief-statement-scenario:
      Belief belief-name belief-description_opt
            scenario-embedded-statement
```

Grammar 5-27 - Scenario beliefs

5.9.3.2 The Scenario Statement

The scenario statement defined in Grammar 5-26 is the only other statement type that has not already been discussed previously. In fact, the scenario statement was already presented in the previous section. However, the scenario statement that is used within scenarios, requires modification so that the trigger and terminate statements can be specified. The modified grammar is defined in Grammar 5-28.

```
scenario-statement:
      scenario-element ;

scenario-element:
      temporal-element
      scenario-element , temporal-element

temporal-element:
      logical-or-element
      atomic { scenario-element }
      parallel { scenario-element }

logical-or-element:
      logical-and-element
      logical-or-element | logical-and-element

logical-and-element:
      one-or-more-element
      logical-and-element & one-or-more-element

one-or-more-element:
      zero-or-more-element
      one-or-more-element +

zero-or-more-element:
      fixed-scenario-element
      zero-or-more-element *

fixed-scenario-element:
      primary-scenario-element
      fixed-scenario-element [ integer-literal ]

primary-scenario-element:
      ( scenario-element )
      scenario-element-embedded-statements

scenario-element-embedded-statements
      scenario-element-embedded-statement
      scenario-element-embedded-statements
            scenario-element-embedded-statement

scenario-element-embedded-statement:
      variable-declaration
      empty-statement
      expression-statement
      belief-statement-scenario
      check-statement
      new-instance-statement
      trigger-statement
      terminate-statement
```

Grammar 5-28 - Modified scenario statement

5.9.3.2.1 The Trigger Statement

The trigger statement is used to indicate the event that creates a new instance of the specified scenario. The trigger statement is formally defined in Grammar 5-29.

```
trigger-statement:
      Trigger ( trigger-element-or )

trigger-element-or:
      trigger-element-and
      trigger-element-or | trigger-element-and

trigger-element-and:
      primary-trigger-element
      trigger-element-or & primary-trigger-element

primary-trigger-element:
      ( trigger-element-or )
      trigger-element-embedded-statement

trigger-element-embedded-statement:
      expression-statement
      new-instance-statement
```

Grammar 5-29 - Trigger statement

As defined in Grammar 5-29 the trigger statement is denoted by the ***Trigger*** keyword followed by a trigger element within parenthesis. The trigger element defines the event that will invoke the scenario. The previously discussed logical or (**|**) and logical and (**&**) operators can be used to connect multiple responsibilities together to define the trigging event. In addition, parenthesis can be used to order responsibilities. Only the expression statement and the new instance statement can be used as a trigger, this translates into an actual instance creation or the invocation of a responsibility as the triggering event.

5.9.3.2.2 The Terminate Statement

The terminate statement performs the opposite task to the trigger statement by indicating the event that terminates the given scenario instance. The definition of the terminate statement is similar to the trigger statement as shown in Grammar 5-30.

```
terminate-statement:
      Terminate ( terminate-element-or )

terminate-element-or:
      terminate-element-and
      terminate-element-or | terminate-element-and

terminate-element-and:
      primary-terminate-element
      terminate-element-or & terminate-trigger-element

primary-terminate-element:
      ( terminate-element-or )
      terminate-element-embedded-statement

terminate-element-embedded-statement:
      expression-statement
      new-instance-statement
```

Grammar 5-30 - Terminate statement

The terminate statement begins with the use of the ***Terminate*** keyword followed by a terminate element within parenthesis. The terminate element follows the same grammar rules as the trigger element. Again, only the expression statement and the new instance statement can be used to terminate a scenario.

5.9.4 Summary

A scenario represents a sequence of events that when executed in a specified order, accomplish a task. Contracts use scenarios to verify execution paths within the IUT. Scenarios are defined by a trigger statement that specifies the event that creates a new scenario instance. The scenario then executes until the event specified by the terminate statement occurs. If the scenario grammar cannot be satisfied the scenario will fail, and a corresponding message will be displayed in the contract evaluation report. Scenarios and responsibilities focus on functional requirements. We will now examine two contract sections that are used to verify non-functional requirements.

5.10 Metric Methods

Metric methods are used to report metrics gathered by the contract during the execution of the IUT. Each metric method returns a specific metric value. Metric values can then be used by metric evaluator plug-ins to analyze the captured metrics. The result of which is written to the contract evaluation report in the reports section. The reports section will be defined in the next section.

A metric method consists of two elements, a definition and a body. The definition of a metric method is defined in Grammar 5-31.

```
metric-declaration:
      metric-modifers_opt Metric return-type identifier
            parameter-list metric-body
      abstract Metric return-type identifier parameter-list ;

metric-modifiers:
      refine

return-type:
      list-or-value_opt return-type-names-no-void

return-type-names-no-void:
      identifier
      built-in-type

parameter-list:
      ( parameters_opt )
      ( Void )

parameters:
      parameter
      parameters , parameter

parameter:
      variable-type identifier

metric-body:
      { metric-statement-list }
```

Grammar 5-31 - Metric method definition

As defined in Grammar 5-31 there are two types of metric method definitions: abstract and regular. Abstract metric method definitions begin with the ***abstract*** modifier followed by the ***Metric*** keyword. The ***Metric*** keyword is used to indicate that a new metric method is being declared. Following the ***Metric*** keyword, a return type must be specified. The return type can either be one of the built-in types or an identifier that is mapped to an IUT type. Each metric method must specify a return type, and the built-in ***Void*** type cannot be used. The rationale here is that the purpose of a metric method is to return a metric value that was gathered during the execution of the IUT. After the return type is specified an identifier is used to denote the name of the metric. Metric method names must be unique within the containing contract. Metric method names must also not have the same name as an invariant, responsibility, scenario, or observability method. Duplicate metric names will result in a compile-time error. Following the metric's name comes an optional parameter set specified between

parenthesis. Each parameter is separated by a comma (**,**). Each parameter consists of two elements, the parameter type and the parameter's name. Parameters can be of either one of the built-in types or as an identifier that is bound to an IUT type. The parameter name is specified via an identifier. Each parameter within a metric method's parameter set must have a unique name. Finally the abstract metric method definition is terminated with a semi-colon (**;**).

Regular metric method definitions begin with an optional modifier. The only legal modifier is the previously discussed ***refine*** modifier. Semantic details regarding the refinement of a metric method will be presented in the next section. Regular metric methods are then declared using the same ***Metric*** keyword, return type, identifier, and optional parameter set. However, instead of terminating the declaration with a semi-colon (**;**), a metric body is specified. Figure 5-12 illustrates an example metric method. The method is defined as a regular metric method, returns an integer value, and accepts a single parameter of type ***T***. The body of the metric method consists of a single statement that will result in an integral value that will be returned to the caller.

```
Metric Integer TimeInContainer(T item)
{
      item_timer.Value(item);
}
```

Figure 5-12 - Metric method example

Metric methods can only be invoked after the IUT has finished executing. As such, a metric method cannot be invoked from within an invariant, responsibility, scenario, or observability method. The only location where a metric method can be invoked is within a reports section or within another metric method. In addition, the body of a metric method cannot invoke methods or query fields within the IUT, as the IUT is no longer running. Such use will result in a compile-time error.

5.10.1 Inheritance Rules

Metric methods defined within a base contract are visible by any contract derived from the base. Metric methods that are marked with the ***abstract*** modifier must be implemented within a derived contract. The contract in which the ***abstract*** modifier is used, must also be marked abstract. If a metric method wishes refine a metric method defined in a base contract, the ***refine*** modifier must be used. The ***refine*** modifier will explicitly indicate that the following metric method is replacing the base method with a matching signature. The refinement will hide the base metric method, and the derived metric method will provide the implementation.

5.10.2 Metric Bodies

Each non-abstract metric method must contain a body. The purpose of the body is to calculate and return the requested metric value. Such calculation can be accomplished by performing mathematical calculations and by invoking other metric methods as defined in Grammar 5-32. Metric methods are prevented from invoking responsibilities, scenarios, and observability methods. However metric methods can access the values of contract variables.

```
metric-statement-list:
      metric-statement
      metric-statement-list metric-statement

metric-statement:
      variable-declaration
      metric-embedded-statement

metric-embedded-statement:
      metric-body
      empty-statement
      expression-statement
      value-statement
```

Grammar 5-32 - Metric body

5.10.2.1 Value Statements

As with observability methods, the metric method's body must return a value. If the body is implemented as a single statement of expression, it is the value of executing the single line that is used as the return value. This is the case in Figure 5-12. Otherwise a value statement must be specified. A value statement is similar to an assignment, where the ***value*** keyword is assigned a value. It is that value that is returned to the caller. For more information on the value statement see the previously discussed observability methods. A metric method whose body is defined as more than a single statement and that does not contain a value statement will generate a compile-time error.

5.11 Reports

Each contract may contain at most one reports section. The reports section is used to invoke the previously discussed metric methods, gather their values and then invoke metric evaluator plug-ins to analyze the metrics, and finally display the result in the contract evaluation report. The reports section is automatically invoked when the execution of the IUT has completed. The ACL does not define any specific metric evaluators; all metric evaluators are provided as plug-ins.

Syntactically a reports section is denoted by the ***Reports*** keyword followed by an open brace (***{***). The body of the reports section is specified between the open brace and a matching close brace (***}***). The close brace denotes the end of the reports section. Grammar 5-33 defines the declaration of the reports section.

```
reports-declaration:
      Reports reports-body

reports-body:
      { reports-statement-list_opt }

reports-statement-list:
      reports-statement
      reports-statement-list reports-statement
```

Grammar 5-33 - Reports declaration

5.11.1 Inheritance Rules

If a base contract contains a reports section, it will be evaluated before the reports section found in the current contract. That is, the flattened contract will contain a single reports section that is composed of the base's reports section followed by the current reports section.

Reports sections cannot be overridden, refined, or hidden through inheritance.

5.11.2 Reports Bodies

As defined in Grammar 5-33, a reports section consists of a body that contains zero or more statements. The statements supported by a reports section are defined in Grammar 5-34.

```
reports-statement:
      variable-declaration
      reports-embedded-statement

reports-embedded-statement:
      reports-body
      empty-statement
      expression-statement
      report-statement
      report-all-statement
```

Grammar 5-34 - Reports body

There are two key types of statements that can occur within the body of a reports section: variable declarations, and embedded statements. Variable declarations are identical to the ones used to declare contract variables, except that the ***exported*** modifier cannot be used. Variables defined within a reports section only exist within the scope of the reports section. Variables defined within a reports

section may have the same name as contract variables. In this case the ***context*** keyword is required in order to access the contract level variables. Variables defined within a reports section must be declared before they are used within the section. That is, a variable declaration must precede its usage.

The second type of statement that can occur within the body of a structure section is an embedded statement. An embedded statement is any statement that is not a variable declaration. In the case of the reports section, general expression statements can be used. Practically, this results in the invocation of metric evaluators, and some arithmetic operations. A reports section can also contain a report and a report all statement.

5.11.2.1 Report Statement

The report statement is used to write to the contract evaluation report. Such writing is usually to report the result of analyzing a metric via a metric evaluator plug-in. The report statement can only be used within a reports section. The report statement is invoked once for each contract instance. Grammar 5-35 defines the report statement.

```
report-statement:
      Report ( string-literal report-parameters_opt ) ;

report-parameters:
      , report-parameter
      report-parameters , report-parameter

report-parameter:
      expression-statement
```

Grammar 5-35 - Report statement

The report statement is similar to the C++ ***printf*** statement, in that a literal string with placeholders is used along with an argument list whose values are used to fill the placeholders. The report statement begins with the ***Report*** keyword followed by parenthesis. Within the parenthesis is a string literal that specifies the string to write the contract evaluation report. Within the string literal placeholders can be used to denote a location where an argument value should be specified. A place holder is denoted by an open brace (***{***) followed by an integer literal indicating the zero-based argument to use, and finally a close brace (***}***) is used to denote the end of the placeholder. If a literal open brace is required, two successive open braces should be specified. Following the string literal is a set of arguments comma (**,**) delimited. The first argument also contains a comma to separate it from the string literal. Each argument is an expression that should result in a value that will be placed into the literal string at the corresponding placeholder location. If the string specifies a placeholder for which a corresponding argument is not specified, a compile-time error will be generated. In addition, a warning will be generated if an argument is specified that is not used. Figure 5-13 provides an example of a report statement.

```
Metric Integer CartSize()
{
      context.cartSize;
}

Reports
{
      Report("The number of items in a cart is: {0}", CartSize());
}
```

Figure 5-13 - Report example

The example shows the ***CartSize()*** metric method being invoked to get a value that will be used to fill the only placeholder in the string literal.

5.11.2.2 ReportAll Statement

The report statement shown in the previous example will display the number of items in each cart once for each instance of the contract that is created. If we only want a single line on the contract evaluation report indicating the average of the cart sizes we can use the report all statement. The report all statement is executed only once per contract, and is able to process metrics gathered by all instances of that contract. Syntactically the report all statement is almost exactly the same as the report statement except for a difference of initial keywords as shown in Grammar 5-36.

```
report-all-statement:
      ReportAll ( string-literal report-parameters_opt ) ;

report-parameters:
      , report-parameter
      report-parameters , report-parameter

report-parameter:
      expression-statement
```

Grammar 5-36 - ReportAll statement

Figure 5-14 adapts the example shown in Figure 5-13 to use the report all statement. Here the ***AvgMetric()*** metric evaluator is used. The metric evaluator is used to calculate average metric values across multiple instances of the same contract. ***AvgMetric()*** takes a single argument that is the number of items that were stored in the customer's cart. The result of executing the ***AvgMetric()*** metric evaluator is used to fill in the placeholder found in the string literal supplied to the report all statement.

```
Metric Integer CartSize()
{
      context.cartSize;
}

Reports
{
      ReportAll("The average number of items in a cart is: {0}",
            AvgMetric(CartSize()));
}
```

Figure 5-14 - ReportAll example

5.11.3 Conclusion

Through the use of metric methods and the reports section, a contract is able to track and analyze metric values gathered while executing the IUT. Metric evaluator plug-ins are used to analyze the gathered metrics. The result of invoking metric evaluators is displayed in the contract evaluation report through the use of the report and the report all statements. We will now look at the final section of a contract: exports.

5.12 Exports

Each contract may have at most one exports section. The exports section is used to denote binding points that are used within the contract. There are three types of binding points: type, method, and field. Type binding points are used to bind an IUT type to a type used within the contract. Method binding points are used to bind an IUT method to a method used within the contract. IUT constructs such as properties or indexers can also be bound using method binding points.[10] Finally, field binding points are used to bind an IUT field to a field within the contract. IUT events can also be bound using field binding points.[11]

In addition to denoting the binding points, the exports section can also be used to specify binding rules for a specific binding point. Binding rules specify restrictions on the set of IUT elements that can be bound to the binding point. Details regarding binding rules will be presented later in this section.

5.12.1 Exports Declaration

An exports section is denoted by the ***Exports*** keyword followed by an open brace (***{***). The body of the exports section is specified between the open brace and a matching close brace (***}***). The close brace denotes the end of the exports section. Grammar 5-37 defines the declaration of the exports section. As shown in Grammar 5-37 there are no modifiers that can be applied to the exports section. In addition the notion of an abstract exports section does not exist within the ACL.

```
exports-declaration:
      Exports exports-body

exports-body:
      { exports-entry-list_opt }

exports-entry-list:
      export-entry
      exports-entry-list export-entry
```

Grammar 5-37 - Exports declaration

5.12.2 Inheritance Rules

If a base contract contains an exports section, all of the binding points defined within the base section will be visible from any derived contracts. In addition, binding points defined within a base contract do not need to be re-specified in a derived contract. If such re-specification occurs, a duplicate binding compile-time error will be generated.

Exports sections cannot be overridden, refined, or hidden through inheritance.

5.12.3 Exports Bodies

As defined in Grammar 5-37, an exports section consists of a body that contains zero or more exports entries. Each export entry corresponds to a binding point. Exports entries are defined in Grammar 5-38.

[10] Such a binding is possible because properties and indexers are represented as specialized methods.
[11] Such a binding is possible because events are represented as specialized fields.

```
export-entry:
      binding-type identifier binding-conformances_opt ;
      binding-type identifier binding-conformances_opt  { binding-rules }

binding-type:
      Type
      Method
      Field

binding-conformances:
      conforms binding-conformance-list

binding-conformance-list:
      identifier export-contract-generics_opt
      binding-conformance-list , identifier export-contract-generics_opt

export-contract-generics:
      < export-generic-parameters >

export-generic-parameters:
      export-generic-parameter
      export-generic-parameters , export-generic-parameter

export-generic-parameter:
      identifier
      literal
```

Grammar 5-38 - Exports body

Each binding entry begins with the type of binding point being created. As previously stated the three types of binding points are types, methods, and fields. The type is specified using the ***Type***, ***Method***, and ***Field*** keywords. Following the binding point type an identifier is specified to denote the symbol that will represent the binding. Each symbol name must be unique and must not conflict with the names of observability methods, invariants, responsibilities, scenarios, or metric methods. If such a conflict occurs, a compile-time error will be generated.

Next, an optional binding conformance may be specified. Binding conformances can only be applied to bindings that are bound to IUT types. That is, binding conformances cannot be applied to methods and fields. A binding conformance indicates a set of contracts that will automatically be bound to the IUT type that is selected for the current binding point. A binding conformance is denoted by the ***conforms*** keyword followed by a list of contracts. Each contract is separated by a comma (**,**). If a contract is generic, appropriate generic parameter values must always be specified.

Following the optional binding conformance is specified, the binding entry can either be terminated by a semi-colon (**;**) or one or more binding rules that are specified between a set of matching braces. Binding rules indicate other bindings within the contract project that must also match the current binding. That is, the IUT element that is bound to the current binding entry must also be bound to the other specified binding entries. Binding rules can also specify that other symbols cannot be bound to the same IUT element. Grammar 5-39 defines the binding rules.

```
binding-rules:
      binding-rule ;
      binding-rules binding-rule ;

binding-rule:
      binding-rule-modifiers_opt scoped-name
      binding-rule-modifiers_opt context

binding-rule-modifiers:
      not
      not derived
      derived

scoped-name:
      identifier
      scoped-name :: identifier
```

Grammar 5-39 - Binding rules

Each binding rule consists of a set of optional modifiers and then either a scoped name or the ***context*** keyword to indicate the current contract's binding. There are two possible modifiers that may be applied: ***not*** and ***derived***.

5.12.3.1 The Not Modifier

The ***not*** modifier is used to indicate that the specified binding symbol cannot be bound to the same IUT element as the containing binding point. By default all binding rules indicate symbols that should be matched to the same IUT element, the ***not*** modifier indicates that the symbols should not be matched.

5.12.3.2 The Derived Modifier

The ***derived*** modifier is used to indicate that not only the specified IUT element, but also any IUT element that is derived from the specified IUT element. By default, derived elements are not taken into account; the ***derived*** modifier indicates that derived elements should be taken into account.

Following the modifiers, a scoped name is used to specify the contract and binding symbol that either should or should not be matched, depending on the use of the ***not*** modifier, to the containing symbol. If the containing contract's IUT type is needed, the ***context*** keyword can be used in place of the scoped name. Figure 5-15 illustrates a few example binding points.

```
Exports
{
	Type T
	{
		not context;
		not derived context;
	}
	Type tFoodItem conforms Item
	{
		Store::tItem;
	}
	Type tCash conforms Cash
	{
		Store::tCash;
	}
	Type tFoodContainer conforms BoundedContainer<tFoodItem, 100>;
	Type tStore conforms Store;
	Field foodContainer tFoodContainer;
}
```

Figure 5-15 - Example exports

5.13 Summary

The central element of the ACL is a contract. Each contract is bound to a type within the IUT. A contract is composed of several elements. Each element has a specific role in the evaluation of the contract. Structure sections are used to specify static checks. A static check is specified via a plug-in that defines a check that can be tested without executing the IUT. Observability methods are query methods that are bound to a method within the IUT. Observability methods are used to capture state information during the execution of the IUT. Invariants are used to describe a set of checks that are tested before and after each responsibility defined within the contract is executed. Responsibilities represent a functional unit of the IUT and are either bound to a method within the IUT or are defined in terms of a grammar of events. Responsibilities are evaluated when their bound IUT counterpart is invoked. Scenarios are used to define a sequence of events that must occur in a specified order during the execution of the IUT. Metric methods are used to report on metrics that were gathered during the execution of the IUT. Reports sections are used to invoke metric evaluators to analyze and report on the metric values retuned by the metric methods. Finally, exports sections are used to specify the binding points required to evaluate the contract, including binding restrictions specified through the use of binding rules.

6 Conclusion

The preceding document has described and defined the ACL in its entirety. The ACL is a high-level contract specification language. The ACL can be used to specify structural checks, responsibilities, scenarios, and metrics that compose a contract. The ACL is an open language which allows for plug-in checks and evaluators to be included within the language.