Module Interface Specification for the Companion Cube Calculator (C^3)

Geneva Smith

December 18, 2017

1 Revision History

Date	Version	Notes
December 18, 2017	2.0	Updated the document to reflect the current implementation
December 17, 2017	1.2	Updated the Control Flow and Input specifications to match the resulting implementation
December 7, 2017	1.1.1	Revised the operator data structure with missing "get" operator, a new exception, and seperated the `numOperands` Integer state variable into three Boolean state variables; added terminator variables to Solver module
December 5, 2017	1.1	Added a specification for an operator data structure
November 27, 2017	1.0	Initial draft completed

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at https://github.com/GenevaS/CAS741/tree/master/Doc/SRS for project symbols, abbreviations, and acronyms.

Contents

3 Introduction

The following document details the Module Interface Specifications for the Companion Cube Calculator (C^3), a mathematical tool which determines the range of a user-specified function given the domains of the function's variables. The calculations are performed using interval arithmetic.

It is assumed that the chosen implementation language will automatically check that the appropriate number of inputs are provided to a function and that all inputs are of the expected type. Therefore, these exceptions are not listed in this specification.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at:

https://github.com/GenevaS/CAS741

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Companion Cube Calculator.

Data Type	**Notation**	**Description**
Boolean	$\mathbb{B}$	The set of $\{True, False\}$
Integer	$\mathbb{Z}$	Any whole number in (-∞, ∞)
Natural	$\mathbb{N}$	Any number in $\{0, 1, 2, 3, ...\}$
Real	$\mathbb{R}$	Any number in (-∞, ∞)
String	$char^n$	A sequence of alphanumeric and special characters

The specification of Companion Cube Calculator uses some derived data types: sequences and strings. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. In addition, Companion Cube Calculator uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project. It can be found at https://github.com/GenevaS/CAS741/blob/master/Doc/Design/MG.

Level 1	**Level 2**
Hardware-Hiding Module	-
Behaviour-Hiding Module	Control Flow Module User Input Module Interval Conversion Module Equation Conversion Module Variable Consolidation Module Range Solver Module Output Module
Software Decision Module	Interval Data Structure Module Equation Data Structure Module Operator Data Structure Module

Table 1: Module Hierarchy

6 MIS of the Control Flow Module

The Control Flow module is the only access point that external applications should use when implementing the Companion Cube Calculator. This affords the freedom to create any type of user interface without changing any of the underlying structure. In some cases, this means that a Control Flow access program simply returns the outputs from other module access programs without modifying them.

6.1 Module

ControlFlow

6.2 Uses

Input (Section 7), Consolidate (Section 10), Solver (Section 11), Output (Section 12), IntervalStruct (Section 13), EquationStruct (Section 14)

6.3 Syntax

6.3.1 Exported Access Programs

Name	In	Out	Exceptions
Initialize	-	$Boolean$	-
ConditionRawInput	$String$, $Boolean$	$String$	-
ControlFile	$String$	$String^n$	-
ControlDirect	$String$, $String$	$String^n$	-
GetSuccessCode	-	Int	-
GetValidFileTypes	-	$String^n$	-
GetDelimiters	-	$String^2$	-
ExtractVariables	$String$	$String^n$	-
GetVariableInfo	-	$String^{n\times 3}$	-

6.4 Semantics

6.4.1 State Variables

- $hasRun : Boolean$
- $successCode : Int$

6.4.2 Access Routine Semantics

Initialize():

- output: *out* := *success* where *success* is the output of the Initialize access program from the Variable Consolidation module
- exception: N/A

ConditionRawInput($input, preserveSpecialChars$):

- output: *out* := *conditionedLine* where *conditioned* is the output of the RemoveWhitespace access program from the Input module
- exception: N/A

ControlFile($fileName$):

- output: *out* := *inputs* where *inputs* is the output of the ReadFile access program from the Input module
- exception: N/A

ControlDirect($equationString, variableListString$):

- transition: Updates the $successCode$ state variable with the return value of the ConvertAndCheckInputs access program from the Consolidate module. If the ControlDirect access program completes successfully, update the *hasRun* state variable to $True$.
- output: *out* := *results* where:
  - The program completed successfully:
    results is the sequence $range, equationTree$. The value for *range* is the output of the PrintInterval access program from the Output module and the value for $equationTree$ is the output of the PrintEquationTree access program from the Output module.
  - The program was not completed successfully, *results* is $NULL$

```
results = NULL
successCode = Consolidate.ConvertAndCheckInputs(
   equationString,
   variableListString,
   Solver.GetValidOperators(),
   Solver.GetValidTerminators())
if successCode = 0
  range = Solver.FindRange(
     Consolidate.GetEquationStruct(),
```

```
            Consolidate.GetIntervalStructList())
        if range != NULL
          results = {Output.PrintInterval(range),
                     Output.PrintEquationTree(Consolidate.
                        GetEquationStruct())}
          hasRun = TRUE
      return results
```

GetSuccessCode():

- output: *out* := *successCode*
- exception: N/A

GetValidFileTypes():

- output: *out* := *validFileTypes* where *validFileTypes* is the output of the GetValidFileTypes access program from the Input module.
- exception: N/A

GetDelimiters():

- output: *out* := *delimiters* where *delimiters* is the set of input delimiters. The sequence size is two, where the first value is the output of the GetLineDelimiter access program from the Input module and the second value is the output of the GetFieldDelimiter access program from the Input module. Both values in the sequence contain unescaped character sequences.
- exception: N/A

ExtractVariables():

- output: *out* := *varList* where *varList* is the output of the ExtractVariablesFromEquation access program from the Consolidate module.
- exception: N/A

GetVariableInfo():

- output: *out* := *varInfoList* where:

```
      Initialize varInfoList = NULL
      if hasRun is TRUE
        intervalList = Consolidate.GetIntervalStructList()
        if the interval list contains data
          foreach(interval in intervalList)
            varInfoList.Add(interval.GetVariableName,
                            interval.GetMinBound,
                            interval.GetMaxBound)
  return varInfoList
```

- exception: N/A

7 MIS of the User Input Module

The Input Module is responsible for the File I/O and string formatting processes required by the program. This module simply outputs a pair of strings (equation and variable information) and leaves the conditioning and validation of the actual values to the Variable Consolidation Module (Section 10). This completely decouples input acquisition from files and the internal function of the program, allowing for other input methods to be implemented simultaneously while reducing the number of modules to modify if the underlying data structures (Section 13 and 14) change.

7.1 Module

Input

7.2 Uses

N/A

7.3 Syntax

7.3.1 Exported Constants

- $lineDelimiter : String$
- $fieldDelimiter : String$
- $validFileTypes : String^n$

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
ReadFile	$String$	$String^2$	IN_CANNOT_READ_FILE, IN_EMPTY_FILE, IN_INVALID_FILE_TYPE, IN_NO_EQUATION, IN_NO_FILE
RemoveWhitespace	$String$, $Bool$	$String$	-
GetLineDelimiter	-	$String$	-
GetFieldDelimiter	-	$String$	-
GetValidFileTypes	-	$String^n$	-

7.4 Semantics

7.4.1 State Variables

N/A

7.4.2 Assumptions

- Input files must be formatted such that:
  - The user equation is on the first line
  - Each subsequent line contains the information (name, minimum bound, and maximum bound) for variables. Each line contains one variable definition, and each field in the variable definition is separated by the $fieldDelimiter$.

  The end of each line must be the value of $lineDelimiter$.

- The conditioning and validation of file contents is performed by the Variable Consolidation module (Section 10).

7.4.3 Access Routine Semantics

ReadFile($fileName$):

- output: $out := fileContents$ where:
  - $fileContents = \{fileName[0], fileName[1, fileName.Length]$ if no exception was raised
  - $fileContents = NULL$ if an exception was raised
- exception: $exc :=$
  $(\neg Read(fileName) \Rightarrow IN\_CANNOT\_READ\_FILE)$
  $|$
  $(Read(fileName) == \emptyset \Rightarrow IN\_EMPTY\_FILE)$
  $|$
  $(fileName.Extension \notin validFileTypes \Rightarrow IN\_INVALID\_FILE\_TYPE)$
  $|$
  $(fileName[0].Exists \wedge fileName[0].Contains(fieldDelimiter) \Rightarrow IN\_NO\_EQUATION)$
  $|$
  $(\neg fileName.Exists \Rightarrow IN\_NO\_FILE)$

RemoveWhitespace($line$, $preserveSpecialWhitespace$):

- output: $out := conditionedLine$ where:
  - If $preserveSpecialWhitespace = TRUE$, $conditionedLine = line$ with white space characters removed except for carriage return (`\r`), line feed (`\n`), and horizontal tab (`\t`) characters
  - If $preserveSpecialWhitespace = FALSE$, $conditionedLine = line$ with all white space characters removed
- exception: N/A

GetLineDelimiter():

- output: $out := lineDelimiter$
- exception: N/A

GetFieldDelimiter():

- output: $out := fieldDelimiter$
- exception: N/A

GetValidFileTypes():

- output: $out := validFileTypes$
- exception: N/A

8 MIS of the Interval Conversion Module

The Interval Conversion module transforms a list of interval data and converts it into a sequence of Interval Data Structures (Section 13). This intermediate step is required so that no modules cannot directly access the conversion process and must go through the Variable Consolidation (Section 10) module instead. This design also ensures that any changes to the input data format or the conversion process will only affect the Variable Consolidation module.

8.1 Module

IntervalConversion

8.2 Uses

IntervalStruct (Section 13)

8.3 Syntax

8.3.1 Exported Access Programs

Name	In	Out	Exceptions
ConvertToIntervals	$String^3$	$IntervalStruct^n$	IVC_CONV_ERR_MAX, IVC_CONV_ERR_MIN, IVC_EMPTY_VARNAME, IVC_MISSING_FIELDS, IVC_NO_BOUNDS, IVC_NO_MAX, IVC_NO_MIN, IVC_TOO_MANY_FIELDS

8.4 Semantics

8.4.1 State Variables

N/A

8.4.2 Assumptions

- Ensuring that $min \leq max$ is handled by the IntervalStruct (Section 13) module.

8.4.3 Access Routine Semantics

ConvertToIntervals($varList, lineDelimiter, fieldDelimiter$):

- output: $out := intervals$ where $intervals$ is a sequence of $IntervalStruct$. First, the input $varList$ is converted into an temporary sequence of strings ($tempSeq$). Then, each element in the $intervals$ sequence is created by taking one of the strings from the temporary sequence and creating an $IntervalStruct$ from its fields, where fields are separated by $fieldDelimiter$.

- exception: $exc :=$
  $(ToReal(max) \notin \mathbb{R} \Rightarrow IVC\_CONV\_ERR\_MAX)$
  $|$
  $(ToReal(min) \notin \mathbb{R} \Rightarrow IVC\_CONV\_ERR\_MIN)$
  $|$
  $(varName = \text{``''} \Rightarrow IVC\_EMPTY\_VARNAME)$
  $|$
  $(\exists l \in tempSeq | l.numFields < 3 \Rightarrow IVC\_MISSING\_FIELDS)$
  $|$
  $(min = max = \text{``''} \Rightarrow IVC\_NO\_BOUNDS)$
  $|$
  $(min \neq \text{``''} \wedge max = \text{``''} \Rightarrow IVC\_NO\_MAX)$
  $|$
  $(min = \text{``''} \wedge max \neq \text{``''} \Rightarrow IVC\_NO\_MIN)$
  $|$
  $(\exists l \in tempSeq | l.numFields > 3 \Rightarrow IVC\_TOO\_MANY\_FIELDS)$

9 MIS of the Equation Conversion Module

The Equation Conversion module creates an equation tree given an equation string using the Precedence Climbing algorithm from:

https://www.engr.mun.ca/ theo/Misc/exp_parsing.htm#climbing

9.1 Module

EquationConversion

9.2 Uses

EquationStruct (Section 14), OperatorStruct (Section 15)

9.3 Syntax

9.3.1 Exported Constants

- $VARTOKEN : String$
- $CONSTTOKEN : String$

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
ConfigureParser	$OperatorStruct^n$, $String^{nx2}$	$Boolean$	EQC_INVALID_OP, EQC_NO_OPS, EQC_UNBALANCED_TERMINATOR
ResetEquationConversion		-	-
MakeEquationTree	$String$	$EquationStruct$	EQC_CONST_FUNC, EQC_IMPLICIT_MULT, EQC_INCOMPLETE_EQ, EQC_INCOMPLETE_OP EQC_UNEXPECTED_TOKEN,
IsReady	-	$Boolean$	-
GetVariableList	-	$String^n$	-
GetVariableToken	-	$String$	-
GetConstToken	-	$String$	-

9.4 Semantics

9.4.1 State Variables

- $ready : Boolean$

- $variableList : String^n$
- $variableStringPattern : String$
- $implicitMultiplicationPattern : String$

9.4.2 Assumptions

- The ConfigureParser function will always be called before any other function in this module.
- The MakeEquationTree function will always be called before the GetVariableList function, otherwise it will not contain any data.

9.4.3 Access Routine Semantics

ConfigureParser($ops, terminators$):

- transition: The value of $variableStringPattern$ and $implicitMultiplicationPattern$ are updated so that operators and terminators are not matched when the module is searching for variables. The symbols from ops are used to update both state variables whereas the symbols from $terminators$ are only used to update $variableStringPattern$. The state variable $ready$ is updated with the output of this access program.
- output: $out := success$ where $success$ is $TRUE$ if no exceptions were encountered and $FALSE$ otherwise.
- exception: $exc :=$
  $(\exists op \in operators | op.IsUnary == False \wedge op.IsBinary == False \Rightarrow EQC\_INVALID\_OP)$
  $|$
  $(operators = \emptyset \Rightarrow EQC\_NO\_OPS)$
  $|$
  $(\exists t[i][2] \in terminators | t[i][2] == "" \Rightarrow EQC\_UNBALANCED\_TERMINATOR)$

ResetEquationConversion():

- transition: All state variables are reset to their default values.
- exception: N/A

MakeEquationTree($equationIn$):

- transition: The value of $variableList$ is updated with new variable names as they are encountered during equation processing.

- output: $out := equationTreeRoot$ where $equationTreeRoot$ is the equation tree produced by the precedence climbing algorithm. This algorithm uses a helper program $Expect(token)$, which raises an exception if the value of $token$ does not match the next character in $equationIn$. Variables in $equationIn$ are recognized using the $variableStringPattern$ state variable.
  Before the algorithm executes, $equationIn$ is conditioned to expand implicit multiplication with an explicit multiplication symbol using $implicitMultiplicationPattern$.

- exception: $exc :=$
  $(ToReal(equationIn) \in \mathbb{R} \Rightarrow EQC\_CONST\_FUNC)$
  $|$
  $(\neg(equationIn.Expand(implicitMultiplicationPattern) = equationIn)$
  $\Rightarrow EQC\_IMPLICIT\_MULT)$
  $|$
  $(\neg(MakeEquationTree.Finish \wedge equationIn = "") \Rightarrow EQC\_INCOMPLETE\_EQ)$
  $|$
  $(\exists op \in equationIn | (NULL < op > userEquation) \vee (userEquation < op > NULL) \Rightarrow$
  $EQC\_INCOMPLETE\_OP)$
  $|$
  $(\neg Expect(token) = token) \Rightarrow EQC\_UNEXPECTED\_TOKEN)$

IsReady():

- output: $out := ready$
- exception: N/A

GetVariableToken():

- output: $out := VARTOKEN$
- exception: N/A

GetConstToken():

- output: $out := CONSTTOKEN$
- exception: N/A

GetVariableList():

- output: $out := variableList$
- exception: N/A

10 MIS of the Variable Consolidation Module

The Variable Consolidation module is responsible for coordinating the conversion process from input values to Equation and Interval Data Structures. This module is intended to be the public interface of the conversion process, and the individual conversion modules should not be accessed directly. This reduces the amount of maintenance required if either the conversion process changes, the format of the inputs change, or any processes that use the results of the conversion process.

10.1 Module

Consolidate

10.2 Uses

IntervalConversion (Section 8), EquationConversion (Section 9), IntervalStruct (Section 13), EquationStruct (Section 14), OperatorStruct (Section 15)

10.3 Syntax

10.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
Initialize	-	$Boolean$	-
ConvertAndCheckInputs	$String$, $String^n$, $OperatorStruct^n$, $String^{n\times 2}$, $String^2$	Int	VC_INIT_FAILED, VC_EXTRA_VARS, VC_MISSING_VARS
ExtractVariablesFromEquation	$String$	$String^n$	-
GetEquationStruct	-	$EquationStruct$	-
GetIntervalStructList	-	$IntervalStruct^n$	-

10.4 Semantics

10.4.1 State Variables

- $equationTreeRoot : EquationStruct$
- $intervalList : IntervalStruct^n$

10.4.2 Access Routine Semantics

Initialize():

- output: *out* := *success* where *success* is the value returned from the ConfigureParser access program from EquationConversion.
- exception: N/A

ConvertAndCheckInputs($eqString$, $varList$, $operators$, $terminators$, $lineDelimiter$, $fieldDelimiter$):

- output: *out* := $successCode$ where $successCode$ can be:
  - 0, if the process completed normally
  - -1, if the ConfigureParser access program from the EquationConversion module failed (The IsReady access program returns $FALSE$)
  - -2, if there are variables in the equation that are not defined in the variable list
  - -3, if the MakeEquationTree access program from the EquationConversion module returned a `NULL` value
- transition:
  - The parameters *operators* and *terminators* are passed as inputs to the ConfigureParser access program from the Equation Conversion module.
  - The state variable $equationTreeRoot$ is assigned the output from the MakeEquationTree access program from the Equation Conversion module with the input parameter $eqString$.
  - The state variable $intervalList$ is assigned the output from the ConvertToIntervals access program from the Interval Conversion module with the input parameters $varList$, $lineDelimiter$, and $fieldDelimiter$.
  - The outputs from the GetVariableList access program from the Equation Conversion module and the GetVariableNamesFromIntervals access program with the input parameter $intervalList$ are compared to ensure that enough variable definitions exist in $intervalList$ to match the existing variables in $equationTreeRoot$.
- exception: $exc :=$
  $(EquationConversion.IsReady == FALSE \Rightarrow VC\_INIT\_FAILED)$
  $|$
  $(\exists var | var \notin eqString \land var \in varList \Rightarrow VC\_EXTRA\_VARS)$
  $|$
  $(\exists var | var \in eqString \land var \notin varList \Rightarrow VC\_MISSING\_VARS)$

ExtractVariablesFromEquation($equation$):

- output: $out := varList$ where $varList$ is the output of the GetVariableList access program after the MakeEquationTree access program has been called. Both access programs are from the EquationConversion module.
- exception: N/A

GetEquationStruct():

- output: $out := equationTreeRoot$
- exception: N/A

GetIntervalStructList():

- output: $out := intervalList$
- exception: N/A

11 MIS of the Range Solver Module

The Range Solver module contains the logic required to perform interval arithmetic operations.

11.1 Module

Solver

11.2 Uses

IntervalStruct (Section 13), EquationStruct (Section 14), OperatorStruct (Section 15)

11.3 Syntax

11.3.1 Exported Constants

- $supportedOps : operatorStruct^n$
- $supportedTerminators : String^{n\text{x}2}$

11.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
GetValidOperators	-	$OperatorStruct^n$	-
GetValidTerminators	-	$String^{n\text{x}2}$	-
FindRange	$EquationStruct$, $IntervalStruct^n$	$IntervalStruct$	SOL_MISSING_VAR, SOL_NO_EQ, SOL_REAL_EXPONENT, SOL_UNSUPPORTED_OP

11.4 Semantics

11.4.1 State Variables

N/A

11.4.2 Assumptions

- The structure of $EquationStruct$ is based on operator precedence.
- The type of $IntervalStruct^n$ accepts `NULL` as a valid value. This supports the computation of constant value equations.

11.4.3 Access Routine Semantics

GetValidOperators():

- output: $out := supportedOps$
- exception: N/A

GetValidTerminators():

- output: $out := supportedTerminators$
- exception: N/A

FindRange($eqRoot, intervals$):

- output: $out := range$ where $range$ is the result of performing and composing interval arithmetic operations on $eqRoot$ using the intervals from $intervals$. The cases are:
  - If $eqRoot$ is a variable, then range is the interval from $intervals$ with that variable name ($varName$)
  - If $eqRoot$ is a constant, then the range is an interval with both bounds set to the constant value
  - If $eqRoot$ is an operator node, then the range is: FindRange($eqRoot.LeftOperand$)$<$ $eqRoot.Operator$ $>$FindRange($eqRoot.RightOperand$)
- exception: $exc :=$
  $(\exists varName \in eqRoot \wedge varName \notin intervals \Rightarrow SOL\_MISSING\_VAR)$
  $|$
  $(eqRoot = NULL \Rightarrow SOL\_NO\_EQ)$
  $|$
  $(\exists op \in eqRoot \wedge op = x^n \wedge n \notin \mathbb{N} \Rightarrow SOL\_REAL\_EXPONENT)$
  $|$
  $((\exists op \in eqRoot \wedge op \notin supportedOps)$
  $\vee (\exists iv1, iv2 \in intervals \wedge \nexists op \in supportedOps | op(iv1, iv2) \vee op(iv2, iv1))$
  $\Rightarrow SOL\_UNSUPPORTED\_OP)$

12 MIS of the Output Module

The Output module is responsible for converting data structures into output-friendly formats.

12.1 Module

Output

12.2 Uses

IntervalStruct (Section 13), EquationStruct (Section 14)

12.3 Syntax

12.3.1 Exported Access Programs

Name	In	Out	Exceptions
PrintInterval	$IntervalStruct$, $Boolean$	$String$	-
PrintEquationTree	$equationStruct$	$String$	-

12.4 Semantics

12.4.1 State Variables

N/A

12.4.2 Assumptions

- There are no exceptions in this module because it is assumed that only well-formed inputs will be passed in. This assumption is made knowing that this module will only be called post-process and any errors in the data structures have already been identified.
- The object passed to PrintEquationTree is the root of the equation tree.

12.4.3 Access Routine Semantics

PrintInterval($interval, withVarName$):

- output: $out := formattedInterval$ where $formattedInterval$ is a string corresponding the fields in $interval$:
  - If $interval$ is a constant value, $formattedInterval$ is $CONST$: with the value appended
  - Otherwise:

  * If $withVarName$ is true, the $formattedInterval$ begins with the interval's variable name and " = "
  * If *interval* is closed on the left boundary, append "[" to $formattedInterval$; otherwise, append "("
  * Append the minimum and maximum boundary values from *interval* separated by a comma (","); if the minimum bound has more that 12 values, put the maximum boundary value on a new line
  * If *interval* is closed on the right boundary, append] to $formattedInterval$; otherwise, append)

- exception: N/A

PrintEquationTree(*eqRoot*):

- output: *out* := $formattedTree$ where $formattedTree$ is a string corresponding the formatted *eqRoot*:

  - If *eqRoot* is a variable, append $+-\{VAR\}$ and the variable name
  - If *eqRoot* is a constant, append $+-\{CONST\}$ and the value
  - If *eqRoot* is an operator node, append $+-\{< op >\}$ where *op* is the node operator:
  - If *eqRoot* is a right operand and has no left of right operands of its own, append " "; this will align the tree levels
  - If *eqRoot* is not a right operator or has a left or right operand, append "|"; this will align the tree levels
  - Append a new line character and print the trees corresponding to the left and right operands of *eqRoot* if they exist

- exception: N/A

13 MIS of the Interval Data Structure Module

The Interval Data Structure represents a mathematical interval with end points *minBound* and *maxBound*. This implementation is designed to be application dependant.

13.1 Module

IntervalStruct

13.2 Uses

N/A

13.3 Syntax

13.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
IntervalStruct	$String$, $\mathbb{R}^2$, $Boolean^2$	$IntervalStruct$	IV_ORD_VIOLATED
GetVariableName	-	$String$	-
GetMinBound	-	$\mathbb{R}$	-
GetMaxBound	-	$\mathbb{R}$	-
IsLeftBoundClosed	-	$Boolean$	-
IsRightBoundClosed	-	$Boolean$	-
SetVariableName	$String$	-	-
SetMinBound	$\mathbb{R}$	-	IV_MIN_ORD_VIOLATED
SetMaxBound	$\mathbb{R}$	-	IV_MAX_ORD_VIOLATED
SetLeftBoundClosed	$Boolean$	-	-
SetRightBoundClosed	$Boolean$	-	-

13.4 Semantics

13.4.1 State Variables

For R2 using DD1

- $variableName : String$
- $minBound : \mathbb{R}$
- $maxBound : \mathbb{R}$
- $isClosedLeft : Boolean$
- $isClosedRight : Boolean$

13.4.2 Access Routine Semantics

IntervalStruct($varName, minB, maxB, leftClosed, RightClosed$):

- output: $out := newIntervalStruct(variableName, minBound, maxBound, isClosedLeft,$ $isClosedRight)$
- transition: Update state variables $variableName$, $minBound$, $maxBound$, $isClosedLeft$, and $isClosedRight$ with the provided values $varName$, $minB$, $maxB$, $leftClosed$, and $rightClosed$
- exception: $exc :=$ $(minB > maxB \Rightarrow IV\_ORD\_VIOLATED)$

GetVariableName():

- output: $out := variableName$
- exception: N/A

GetMinBound():

- output: $out := minBound$
- exception: N/A

GetMaxBound():

- output: $out := maxBound$
- exception: N/A

IsLeftBoundClosed():

- output: $out := isClosedLeft$
- exception: N/A

IsRightBoundClosed():

- output: $out := isClosedRight$
- exception: N/A

SetVariableName($varName$):

- transition: Update state variable $variableName$ with the provided value $varName$
- exception: N/A

SetMinBound($minB$):

- transition: Update state variable $minBound$ with the provided value $minB$
- exception: $exc :=$
  $(minB > maxBound \Rightarrow IV\_MIN\_ORD\_VIOLATED)$

SetMaxBound($maxB$):

- transition: Update state variable $maxBound$ with the provided value $maxB$
- exception: $exc :=$
  $(maxB < minBound \Rightarrow IV\_MAX\_ORD\_VIOLATED)$

SetLeftBoundClosed($closed$):

- transition: Update state variable $isClosedLeft$ with the provided value $closed$
- exception: N/A

SetRightBoundClosed($closed$):

- transition: Update state variable $isClosedRight$ with the provided value $closed$
- exception: N/A

14 MIS of the Equation Data Structure Module

The Equation Data Structure represents a node in an equation tree which can support up to two-operand operations. The tree can be expanded by assigning other nodes as the left and right operands.

14.1 Module

EquationStruct

14.2 Uses

N/A

14.3 Syntax

14.3.1 Exported Access Programs

Name	In	Out	Exceptions
EquationStruct	$String^2$, $EquationStruct^2$	$EquationStruct$	EQS_MISSING_OP
GetOperator	-	$String$	-
GetVariableName	-	$String$	-
GetLeftOperand	-	$EquationStruct$	-
GetRightOperand	-	$EquationStruct$	-
SetVariableName	$String$	-	-
SetLeftOperand	$EquationStruct$	-	-
SetRightOperand	$EquationStruct$	-	-

14.4 Semantics

14.4.1 State Variables

To support R4 and R6

- $operatr : String$
- $variableName : String$
- $leftOperand : EquationStruct$
- $rightOperand : EquationStruct$

14.4.2 Assumptions

- The decomposition of the user equation is handled by the Equation Conversion module (Section 9).
- Unsupported operators are identified and handled in the Equation Conversion module (Section 9).
- There is no setter method for the *operator* field because they will not be changed after initialization.
- The values for $leftOperand$ and $rightOperand$ can be set to `NULL` as required (e.g. variables, constants).

14.4.3 Access Routine Semantics

EquationStruct($op, vName, eStruct1, eStruct2$):

- output: $out := newEquationStruct(operatr, variableName, leftOperand, rightOperand)$
- transition: Update state variables $operatr$, $variableName$, $leftOperand$, and $rightOperand$ with the provided values op, $vName$, $eStruct1$, and $eStruct2$
- exception: $exc :=$
  $(op = \text{``''} \Rightarrow EQS\_MISSING\_OP)$

GetOperator():

- output: $out := operatr$
- exception: N/A

GetVariableName():

- output: $out := variableName$
- exception: N/A

GetLeftOperand():

- output: $out := leftOperand$
- exception: N/A

GetRightOperand():

- output: $out := rightOperand$
- exception: N/A

SetVariableName($vName$):

- transition: Update state variable $variableName$ with the provided value $vName$
- exception: N/A

SetLeftOperand($eStruct$):

- transition: Update state variable $leftOperand$ with the provided value $eStruct$
- exception: N/A

SetRightOperand($eStruct$):

- transition: Update state variable $rightOperand$ with the provided value $eStruct$
- exception: N/A

15 MIS of the Operator Data Structure Module

The Operator Data Structure contains all relevant information required to correctly use them in a mathematical context. It is much simpler to pass a single data structure containing all of the associated fields for an operator as opposed to creating a class with lists of information that must queried and returned individually for each associated operator field.

15.1 Module

OperatorStruct

15.2 Uses

N/A

15.3 Syntax

15.3.1 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
OperatorStruct	$String$, Int, $Boolean$[4]	$OperatorStruct$	OP_INVALID_PRECEDENCE, OP_MISSING_OP, OP_MULTI_TYPE, OP_NO_TYPE
GetOperator	-	$String$	-
GetPrecedence	-	Int	-
IsUnary	-	$Boolean$	-
IsBinary	-	$Boolean$	-
IsTernary	-	$Boolean$	-
IsLeftAssociative	-	$Boolean$	-

15.4 Semantics

15.4.1 State Variables

- $operatr : String$
- $precedence : Int$
- $isUnary : Boolean$
- $isBinary : Boolean$
- $isTernary : Boolean$
- $leftAssociative : Boolean$

15.4.2 Assumptions

- There are no Setter methods for this module because operator properties are fixed.
- A high integer value is associated with a high precedence operation.

15.4.3 Access Routine Semantics

OperatorStruct($op, prec, isUnary, isBinary, isTernary, isLeftAssociative$):

- output: $out := newOperatorStruct(operatr, precedence, isUnary, isBinary, isTernary, leftAssociative)$
- transition: Update state variables $operatr$, $precedence$, $isUnary$, $isBinary$, $isTernary$, and $leftAssociative$ with the provided values op, $prec$, $isUnary$, $isBinary$, $isTernary$, and $isLeftAssociative$.
- exception: $exc :=$
  $(prec < 0 \Rightarrow OP\_INVALID\_PRECEDENCE)$
  $|$
  $(op = \text{“”} \Rightarrow OP\_MISSING\_OP)$
  $|$
  $((isUnary = isBinary \wedge isUnary = True) \vee (isUnary = isTernary \wedge isUnary = True) \vee (isBinary = isTernary \wedge isBinary = True) \Rightarrow OP\_MULTI\_TYPE)$
  $|$
  $(isUnary = isBinary = isTernary \wedge isUnary = False \Rightarrow OP\_NO\_TYPE)$

GetOperator():

- output: $out := operatr$
- exception: N/A

GetPrecedence():

- output: $out := precedence$
- exception: N/A

IsUnary():

- output: $out := isUnary$
- exception: N/A

IsBinary():

- output: $out := isBinary$

- exception: N/A

IsTernary():

- output: $out := isTernary$
- exception: N/A

IsLeftAssociative():

- output: $out := leftAssociative$
- exception: N/A

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

16 Appendix

Table 2: Possible Error Exceptions

Message ID	Error Message
EQC_INVALID_OP	Error: The equation conversion module cannot parse the $< token >$ operator.
EQC_NO_OPS	Error: No operators were passed to the parser.
EQC_UNBALANCED_TERMINATOR	Error: An unbalanced (left |right) terminator token was encountered ($< token >$).
EQC_INCOMPLETE_EQ	Error: Could not find the end of the equation.
EQC_INCOMPLETE_OP	Error: Unrecognized sequence encountered during Atomic Equation parsing. Remaining equation = $< remainingEquationString >$.
EQC_UNEXPECTED_TOKEN	Error: Could not find expected token $< token >$.
EQS_MISSING_OP	Error: Equation structures must be assigned an operator during initialization.
IN_CANNOT_READ_FILE	Error: The file could not be read.
IN_EMPTY_FILE	Error: The file is empty.
IN_INVALID_FILE_TYPE	Error: Cannot read files of this type.
IN_NO_EQUATION	Error: The first line of the file is not an equation or the equation contains $< Input.GetFieldDelimiter >$.
IN_NO_FILE	Error: The specified file does not exist.
IVC_CONV_ERR_MIN	Error: The string provided for the minimum bound cannot be converted to a real number.
IVC_CONV_ERR_MAX	Error: The string provided for the maximum bound cannot be converted to a real number.
IVC_EMPTY_VARNAME	Error: Intervals must have an associated variable name.
IVC_MISSING_FIELDS	Error: No fields found for variable (Line $< lineNumber >$). Skipping line.
IVC_NO_BOUNDS	Error: No values provided for either interval bound.
IVC_TOO_MANY_FIELDS	Error: Encountered a variable with more than three fields (Line $< lineNumber >$). Skipping line.
OP_INVALID_PRECEDENCE	Error: Cannot assign a precedence value less than 0.
OP_MISSING_OP	Error: Cannot have an operator with no representative symbol.

<table>
<tr><td>OP_MULTI_TYPE</td><td>Error: An operator cannot be overloaded to be unary, binary, and ternary.</td></tr>
<tr><td>OP_NO_TYPE</td><td>Error: Operators must be assigned a number of operands type.</td></tr>
<tr><td>SOL_MISSING_VAR</td><td>Error: Could not find an associated interval for variable $< \mathit{varname} >$.</td></tr>
<tr><td>SOL_NO_EQ</td><td>Error: No information was provided for the equation.</td></tr>
<tr><td>SOL_UNSUPPORTED_OP</td><td>Error: An unsupported operation was encountered while solving for the range of the equation (Unknown operator |Mixed interval division |Exponents |Exponent base $<= 1$ |Exponent < 0).</td></tr>
<tr><td>VC_INIT_FAILED</td><td>Error: Equation parser could not be configured.</td></tr>
<tr><td>VC_MISSING_VARS</td><td>Error: Cannot find intervals for variable name(s): $< \mathit{variablelist} >$.</td></tr>
</table>

Table 3: Possible Warning Exceptions

Message ID	**Error Message**
EQC_CONST_FUNC	Warning: The user equation is a constant value and the range will only include this value.
EQC_IMPLICIT_MULT	Warning: Encountered an implicit multiplication of a constant value and a variable. Expanding with explicit operator.
IV_MAX_ORD_VIOLATED	Warning: Value provided for maximum bound is smaller than the current minimum bound. The values have been exchanged to maintain the interval ordering.
IV_MIN_ORD_VIOLATED	Warning: Value provided for minimum bound is greater than the current maximum bound. The values have been exchanged to maintain the interval ordering.
IV_ORD_VIOLATED	Warning: Value provided for intervals are not in increasing order. The values have been exchanged to maintain the interval ordering.
IVC_NO_MIN	Warning: No minimum interval bound given. Setting it to the same value as the maximum bound.
IVC_NO_MAX	Warning: No maximum interval bound given. Setting it to the same value as the minimum bound.
SOL_REAL_EXPONENT	Warning: The value provided for the exponent $< N >$ is not a natural number. It has been rounded to $< Round(N) >$.
VC_EXTRA_VARS	Warning: Extraneous variables found in interval list $(< variablelist >)$.