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

Geneva Smith

November 23, 2017

1 Revision History

Date	Version	Notes
	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.

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

[You should describe your notation. You can use what is below as a starting point. —SS]

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
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

The specification of Companion Cube Calculator uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. 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

Table 1: Module Hierarchy

6 MIS of the Control Flow Module

6.1 Module

main

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
main	-	-	-

6.4 Semantics

6.4.1 State Variables

N/A

6.4.2 Access Routine Semantics

main():

- transition: Create data structures to contain the user inputs and modify their states for the Output module.

```
# Get User Input
  eqString := Input.getUserEquation()
  varList := Input.getVariableList()

# Convert input into equation and interval data structures using
# the list of valid operators from Solver
  operators := Solver.getValidOperators()
  Consolidate.convertAndCheckInputs(eqString, varList, operators)

# Get the equation and interval data structures and pass them to
# the Solver module
  equationData := Consolidate.getEquationStruct()
  intervalDataList := Consolidate.GetIntervalStructList()
  range := Solver.calculateRange(equationData, intervalDataList)
```

```
# Report the results back to the user
  Output.printInterval(range)
  Output.printEquationTree(equationData)
  for each interval in intervalDataList:
    Output.printInterval(interval)
```

7 MIS of the User Input Module

7.1 Module

Input

7.2 Uses

7.3 Syntax

7.3.1 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

7.4 Semantics

7.4.1 State Variables

7.4.2 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

8 MIS of the Interval Conversion Module

8.1 Module

[Short name for the module —SS]

8.2 Uses

8.3 Syntax

8.3.1 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

8.4 Semantics

8.4.1 State Variables

8.4.2 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

9 MIS of the Equation Conversion Module

9.1 Module

[Short name for the module —SS]

9.2 Uses

9.3 Syntax

9.3.1 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

9.4 Semantics

9.4.1 State Variables

9.4.2 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

10 MIS of the Variable Consolidation Module

10.1 Module

Consolidate

10.2 Uses

10.3 Syntax

10.3.1 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

10.4 Semantics

10.4.1 State Variables

10.4.2 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

11 MIS of the Range Solver Module

11.1 Module

Solver

11.2 Uses

11.3 Syntax

11.3.1 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

11.4 Semantics

11.4.1 State Variables

11.4.2 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]
- exception: [if appropriate —SS]

12 MIS of the Output Module

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*	-	-
printEquationTree	*equationStruct*	-	-

12.4 Semantics

12.4.1 State Variables

N/A

12.4.2 Environment Variables

- cmd: the command-line interface
- win: a 2D sequence of pixels displayed on the screen

12.4.3 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

12.4.4 Access Routine Semantics

printIntervalList(*intervalStruct*):

- transition: If the user interface is the command-line, write the interval *intervalStruct* to cmd. If the user interface is a GUI, modify win so that the interval is displayed. In both cases, the variable name of the interval must also be displayed.

- exception: N/A

printEquationTree(*equationStruct*):

- transition: If the user interface is the command-line, write the equation tree to cmd. If the user interface is a GUI, modify win so that the equation tree is displayed.
- exception: N/A

13 MIS of the Interval Data Structure Module

13.1 Module

intervalStruct

13.2 Uses

N/A

13.3 Syntax

13.3.1 Exported Access Programs

Name	In	Out	Exceptions
intervalStruct	$\mathbb{R}^2$	*intervalStruct*	IV_NOT_REAL_TYPE, IV_INSUFF_PARAMS, ORD_VIOLATED
GetMinRange	-	$\mathbb{R}$	-
GetMaxRange	-	$\mathbb{R}$	-
SetMinRange	$\mathbb{R}$	-	IV_NOT_REAL_TYPE, IV_INSUFF_PARAMS, ORD_VIOLATED
SetMaxRange	$\mathbb{R}$	-	IV_NOT_REAL_TYPE, IV_INSUFF_PARAMS, ORD_VIOLATED

13.4 Semantics

13.4.1 State Variables

For R2 using DD1

- $a : \mathbb{R}$
- $b : \mathbb{R}$

13.4.2 Access Routine Semantics

intervalStruct($val1, val2$):

- output: $out := intervalStruct$
- transition: Update state variables a and b with the provided values $val1$ and $val2$

- exception: $exc :=$
  $(val1 \notin \mathbb{R} \vee val2 \notin \mathbb{R} \Rightarrow IV\_NOT\_REAL\_TYPE)$
  |
  $(\nexists val1 \vee \nexists val2 \Rightarrow IV\_INSUFF\_PARAMS)$
  |
  $(val1 > val2 \Rightarrow ORD\_VIOLATED)$

intervalStruct.GetMinRange():

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

intervalStruct.GetMaxRange():

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

intervalStruct.SetMinRange(val):

- transition: Update state variable a with the provided value val
- exception: $exc :=$
  $(val \notin \mathbb{R} \Rightarrow IV\_NOT\_REAL\_TYPE)$
  |
  $(\nexists val \Rightarrow IV\_INSUFF\_PARAMS)$
  |
  $(val > b \Rightarrow ORD\_VIOLATED)$

intervalStruct.SetMaxRange(val):

- transition: Update state variable b with the provided value val
- exception: $exc :=$
  $(val \notin \mathbb{R} \Rightarrow IV\_NOT\_REAL\_TYPE)$
  |
  $(\nexists val \Rightarrow IV\_INSUFF\_PARAMS)$
  |
  $(val < a \Rightarrow ORD\_VIOLATED)$

14 MIS of the Equation Data Structure Module

14.1 Module

equationStruct

14.2 Uses

N/A

14.3 Syntax

14.3.1 Exported Access Programs

Name	In	Out	Exceptions
equationStruct	String, *equationStruct*[2]	*equationStruct*	EQ_INSUFF_PARAMS, EQ_WRONG_OPERATOR_TYPE, EQ_WRONG_OPERAND_TYPE
GetOperator	-	String	-
GetLeftOperand	-	*equationStruct*	-
GetRightOperand	-	*equationStruct*	-
SetLeftOperand	*equationStruct*	-	EQ_INSUFF_PARAMS, EQ_WRONG_OPERAND_TYPE
SetRightOperand	*equationStruct*	-	EQ_INSUFF_PARAMS, EQ_WRONG_OPERAND_TYPE

14.4 Semantics

14.4.1 State Variables

To support R4 and R6

- op : string
- x : *equationStruct*
- y : *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 *op* field because it will not be changed after initialization.
- The values for x and y can be set to `NULL` as required.

14.4.3 Access Routine Semantics

equationStruct($operator, eStruct1, eStruct2$):

- output: $out := equationStruct$
- transition: Update state variables op, x, and y with the provided values $operator$, $eStruct1$, and $eStruct2$
- exception: $exc :=$
  $(\nexists operator \lor \nexists eStruct1 \lor \nexists eStruct2 \Rightarrow EQ\_INSUFF\_PARAMS)$
  |
  $(operator \neq string \Rightarrow EQ\_WRONG\_OPERATOR\_TYPE)$
  |
  $(eStruct1 \neq equationStruct \lor eStruct2 \neq equationStruct$
  $\Rightarrow EQ\_WRONG\_OPERAND\_TYPE)$

equationStruct.GetOperator():

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

equationStruct.GetLeftOperand():

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

equationStruct.GetRightOperand():

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

SetLeftOperand($eStruct$):

- transition: Update state variable x with the provided value $eStruct$
- exception: $exc :=$
  $(\nexists eStruct \Rightarrow EQ\_INSUFF\_PARAMS)$
  |
  $(eStruct \neq equationStruct \Rightarrow EQ\_WRONG\_OPERAND\_TYPE)$

SetRightOperand($eStruct$):

- transition: Update state variable y with the provided value $eStruct$
- exception: $exc :=$
  $(\nexists eStruct \Rightarrow EQ\_INSUFF\_PARAMS)$
  |
  $(eStruct \neq equationStruct \Rightarrow EQ\_WRONG\_OPERAND\_TYPE)$

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.

15 Appendix

Table 2: Possible Exceptions

Message ID	**Error Message**
EQ_INSUFF_PARAMS	Error: Insufficient number of parameters provided to equation data structure.
EQ_WRONG_OPERAND_TYPE	Error: Operands must have type *equationStruct*.
EQ_WRONG_OPERATOR_TYPE	Warning: The operator must have type *string*. String type conversion has been applied.
IV_NOT_REAL_TYPE	Error: Interval values must be of type real.
IV_INSUFF_PARAMS	Error: Insufficient number of parameters provided to interval data structure.
ORD_VIOLATED	Warning: Value provided for intervals are not in increasing order. The values have been exchanged to maintain the interval ordering.