Module Interface Specification for ...

Author Name

November 4, 2019

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at [give url —SS]

[Also add any additional symbols, abbreviations or acronyms —SS]

Contents

3 Introduction

The following document details the Module Interface Specifications for [Fill in your project name and description —SS]

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at .... [provide the url for your repo —SS]

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 FSL.

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 FSL 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, FSL 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.

Level 1	**Level 2**
Hardware-Hiding	
Behaviour-Hiding	Input Parameters Output Format Output Verification Temperature ODEs Energy Equations Control Module Specification Parameters Module
Software Decision	Sequence Data Structure ODE Solver Plotting

Table 1: Module Hierarchy

6 MIS of [Module Name —SS]

[Use labels for cross-referencing —SS]

[You can reference SRS labels, such as R??. —SS]

[It is also possible to use LaTeXfor hypperlinks to external documents. —SS]

6.1 Module

[Short name for the module —SS]

6.2 Uses

6.3 Syntax

6.3.1 Exported Constants

None.

6.3.2 Exported Data Type

There is one exported data type, `CFST`, which is the type of a CFS object. (abstract, with one type template `FLOAT`, indicating floating point types) The structure of `CFST` is

- n: integer
- ω: `FLOAT`
- A: sequence of `FLOAT`
- B: sequence of `FLOAT`

During implementation, it is recommended that A and B are random accessible.

6.3.3 Exported Access Programs

Name	In	Out	Exceptions
`ConvertFrom`	n: integer ω: `FLOAT` A: sequence of `FLOAT` B: sequence of `FLOAT`	Constructed CFS	Mismatch between n, (size of A -1), and size of B
`ConvertTo`	CFS pointers to the space for outputs	Same as inputs of `ConvertFrom`	None
`TransformTo`	function f, n, ω	Constructed CFS	Any exception raised by integral module
`FunctionValue`	CFS, t	Value of $f(t)$, where CFS comes from f	None.
`Addition`	`CFST CFS1, CFST CFS2`	`CFST, CFSres`	mismatch of n and ω
`Subtraction`	`CFST CFS1, CFST CFS2`	`CFST, CFSres`	Same as `Addition`
`Multplication`	`CFST CFS1, CFST CFS2`	`CFST, CFSres`	Same as `Addition`
`Divison`	`CFST CFS1, CFST CFS2`	`CFST, CFSres`	Same as `Addition`, plus any exception raised by linear solver and reclassified in `Division`.
`ToleratedEquality`	`CFST CFS1, CFST CFS2, FLOAT tol`	`Bool res`	Same as `Addition`
`Function`	`CFST CFS`, Taylor series type variable	`CFST, CFSres`	Insufficient length of Taylor series.
`Amplitude`	CFS	`FLOAT amp`	None.

6.4 Semantics

6.4.1 State Variables

[Not all modules will have state variables. State variables give the module a memory. —SS]

6.4.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

6.4.3 Assumptions

[Try to minimize assumptions and anticipate programmer errors via exceptions, but for practical purposes assumptions are sometimes appropriate. —SS]

6.4.4 Access Routine Semantics

[accessProg —SS]():

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

[A module without environment variables or state variables is unlikely to have a state transition. In this case a state transition can only occur if the module is changing the state of another module. —SS]

[Modules rarely have both a transition and an output. In most cases you will have one or the other. —SS]

6.4.5 Local Functions

[As appropriate —SS] [These functions are for the purpose of specification. They are not necessarily something that is going to be implemented explicitly. Even if they are implemented, they are not exported; they only have local scope. —SS]

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.

7 Appendix

[Extra information if required —SS]