Designing and Evaluating a Configuration DSL for Ocean Models

Serafim Simonov

November 17, 2020
Motivation

- Simulations are based on mathematical models
- Development requires knowledge in different domains
  - Mathematics
  - Natural sciences
  - Software development
  - Parallelism
- Solution: DSL
Goals

- Analyze Domain Problem
- DSL Development
- Evaluation
Foundations and Technologies

Foundations

- DSL Development Approach Mernik et al. [2005]
- GECO Approach [Jung 2016]
- DSL Framework XText [Efftinge et al. 2012]
- Language Server Protocol
MIT General Circulation Model

- Case Study
- Docker Container
- Evaluation
Plausibility Check Process

Domain Analysis

- Configure Compile-Time Model Settings
- Compile
- Setup Parameters for Test Scenarios
- Run Test Scenarios
- Analyse Test Results

Tests where successful

no, wrong compile-time settings

no, wrong parameter settings
Deployment and Execution Process

Domain Analysis

1. Configure Compile-Time Model Settings
2. Compile
3. Parameter Setup
4. Scenario Study
5. Sensitivity Study
6. Run Scenario Study
7. Run Sensitivity Study

- Compilation successful
- Type of study
- Scenario study
- Sensitivity study
- yes
- no
- setup Parameters with Variations

Serafim Simonov
Designing and Evaluating a Configuration DSL
November 17, 2020 7 / 20
Configuration Files of MITGcm

Domain Analysis

- SIZE.H
- eedata
- data
- Optional Files
Implementation Overview

- Megamodel
  - Grammar
  - Metamodels
- Unit Parser
- Validator
- Generator
Megamodel

Implementation

ParameterDeclaration

references

Units Meta Model

Type System

references

DSL Grammar
Grammar

Implementation

Model:
  mainblock=MainBody;

MainBody:
  name=ID ':' model=[declaration::DeclarationModel|ID] '{'
  (paramGroups+=ParamGroup)*
  '}'

ParamGroup:
  name=ID '{'
  (params+=Param)*
  '}'

Param:
  declaration=[declaration::ParamDeclaration|ID] '=' value=Literal (unit=Unit)?;
Grammar

Implementation

Model:
  mainblock=MainBody;

MainBody:
  name=ID ': ' model=[declaration::DeclarationModel|ID] '{'
  (paramGroups+=ParamGroup)*
  '}'

ParamGroup:
  name=ID '{'
  (params+=Param)*
  '}'

Param:
  declaration=[declaration::ParamDeclaration|ID] '=' value=Literal (unit=Unit)?;
Grammar

Implementation

```plaintext
Model:
    mainblock=MainBody;

MainBody:
    name=ID ':' model=[declaration::DeclarationModel|ID] '{'
        (paramGroups+=ParamGroup)*
    '}'

ParamGroup:
    name=ID '{'
        (params+=Param)*
    '}'

Param:
    declaration=[declaration::ParamDeclaration|ID] '=' value=Literal (unit=Unit)?
```
Grammar

Implementation

Model:

mainblock=MainBody;

MainBody:

name=ID ':' model=[declaration::DeclarationModel|ID] '{'
(paramGroups+=ParamGroup)*
'}';

ParamGroup:

name=ID '{' (params+=Param)*
'}';

Param:

declaration=[declaration::ParamDeclaration|ID] '!=' value=Literal (unit=Unit)?;
Metamodel Type System

Implementation

TypeModel

Type

name: String

TypeReference

PrimitiveType

ArrayType
Units Metamodel

Implementation

- EPrefix
- PrimitiveUnit
- DividedUnit
- ExponentUnit
- MultipleUnits
- ESIUnitType
- SIUnit
- CustomUnit
Unit Parser

Implementation

C

unambiguous unit

q_0 → c

a → c

b/c → c

m → mo

o → mol

cca/cPa → a

End
cm

Diagram showing a flowchart with nodes labeled `q_0`, `c`, `ca`, `cc/cP`, `cca/cPa`, `m`, `mo`, and `mol`, with arrows indicating the flow between them. The `unambiguous unit` concept is indicated by a note.
cmo

Diagram:
- **End**
- **q_0** connected to **c**
- **c** connected to **m**
- **m** connected to **mo**
- **mo** connected to **mol**
- **cc/cP** connected to **cca/cPa**
- **ca** connected to **c/P**
- **a** connected to **c/P**
- **unambiguous unit**

Nodes:**
- **End**
- **q_0**
- **c**
- **m**
- **mo**
- **mol**
- **cc/cP**
- **cca/cPa**
- **ca**
- **c/P**
- **a**
cmol

Diagram:
- `cmol` starts with `q_0` and goes through `c`, `m`, `mo`, and finally `mol`.
- `c` has transitions to `ca` and `cc/cP`.
- `ca` has a transition to `cc/cP`.
- `cc/cP` has a transition to `cca/cPa`.
- `mmo` has a transition to `mol`.
- The diagram includes an unambiguous unit with transitions labeled `c`, `a`, and `c/P`.

This diagram represents the configuration of the `cmol` unit in a system, showing the flow of different components and their interactions.
Validator

Implementation

- Type check
- Unit check
- Check of parameter constraints
Implementation provided for *data*, *SIZE.h*, *eedata*

Strategy Pattern

```
MITGcmGeneratorContext
  - fsa:FileSystemAccess2
  - params: BasicEList
  + generate()

<<Interface>>
  1..*
  MITGcm
    generate(): CharSequence
    getName(): String

SizeHGenerator
  params: BasicEList

DataGenerator
  params: BasicEList

EeDataGenerator
  params: BasicEList
```
Evaluation

Evaluation criteria

- Can we use the DSL to specify proper configurations?
- Can the generator produce MITgcm configurations?
- Is the DSL usable and understandable by a user?
Evaluation

Evaluation setup

- JupyterLab using LSP
- Docker Build for MITgcm
- Three example experiments

Evaluation result

- Model run produces the same output
- Some minor formatting issues
- Test users wish better context help
Conclusion

- Concept works with limitations
- Compatible configuration files
- User experience must be improved

Future Work

- Import support
- Implement generator for all configuration files
- Study another Ocean Models
- Professional advice