Verifying ParamGen

A Case Study in Scientific Software Abstraction and Modeling

Alper Altuntas[1], John Baugh[2]

NCAR
UCAR

[1]NCAR/CGD
[2]NCSU

Improving Scientific Software Conference, April 17, 2023, Boulder, CO

Formal Specification

Describing software using a precise and high-level language.

- Helps design maintainable and reliable software.

What is ParamGen?

An infrastructure library that generates default input parameter files for ESM components.

- Similar to `CIME.nmlgen`
- Supports any format (not just `nml`)
- Supports any template format (not just `xml`)
- Supports arbitrary Python expressions in template files!

What is ParamGen?

An infrastructure library that generates default input parameter files for ESM components.

- Similar to `CIME.nmlgen`
- Supports any format (not just `nml`)
- Supports any template format (not just `xml`)
- Supports arbitrary Python expressions in template files!

- Initially developed for MOM6 in CESM.
- Used by MOM6, CAM, NUOPC, and WW3 of CESM.

What is ParamGen?

High-level configs

```
OCN_GRID = "t061"
```

Template

```
NIHALO: 2

NIGLOBAL:
  $OCN_GRID == "g16":
    320
  $OCN_GRID == "t061":
    540

DIABATIC_FIRST:
  $OCN_GRID in ["g16","t061"]
```

```
ParamGen.reduce()
```

Model input file

```
NIHALO = 2

NIGLOBAL = 540

DIABATIC_FIRST = True
```

ParamGen Terminology

↓ *branch*

label →
guard →
value →

```
NIGLOBAL:
  $OCN_GRID == "g16":
    320
```

expandable variable →

```
  $OCN_GRID
```

ParamGen is flexible!

- Expandable variables and Python expressions can appear in labels, guards, values.
- There can be multiple labels along a branch.
- There can be multiple (nested) guards along a branch.
- No restriction on the ordering of guards and labels.

ParamGen schema!

- **Schema**:
  - There must be at least one label along a branch.
  - If a key is a guard, all its siblings must also be guards.

invalid!

```
NIGLOBAL:
   description:
     "grid points in x-dir."
   $OCN_GRID == "g16":
     320
```

valid

```
NIGLOBAL:
   description:
      "grid points in x-dir."
   value:
      $OCN_GRID == "g16":
         320
```

ParamGen `reduce()` method

```
def reduce(data):

  if isinstance(data, dict):
    # (1) Expand vars in keys
    data = expand_key_vars(data)

    # (2) Evaulate guards (if applicable)
    if is_guarded_dict(data):
      data = impose_guards(data))

    # (3) Call reduce recursively
    else:
      for key in data:
        data[key] = reduce(data[key])

  else:
    # (4) expand vars, apply formulas
    data = expand_vars(data)

  return data
```

- Traverse all branches recursively.
- Apply in-place modifications:
  - expand variables.
  - evaluate formulas.
  - impose guards.

How to ensure reliability?

Sources of complexity

- Recursive algorithm with in-place data modifications.
- Great flexibility in template layout.

How to ensure reliability?

Sources of complexity

- Recursive algorithm with in-place data modifications.
- Great flexibility in template layout.

Testing

- Routine unit and integration tests.

How to ensure reliability?

Sources of complexity

- Recursive algorithm with in-place data modifications.
- Great flexibility in template layout.

Testing

- Routine unit and integration tests.

"Testing can be used to show the presence of bugs, but never to show their absence!."

— Edsger W. Dijkstra

Proving programs correct

Formal methods: mathematically rigorous techniques and tools for the specification, design and verification of software. (Butler, R. W., 2001)

Proving programs correct

Formal methods: mathematically rigorous techniques and tools for the specification, design and verification of software. (Butler, R. W., 2001)

Alloy: an open source language and tool for software modeling.

- The Alloy language is simple, precise, powerful, and elegant!
- This allows us to model software and algorithms at a high-level. (at the design level, and above the code level)

The Alloy Language

in a few slides

Alloy

A software modeling and analysis tool with a **declarative** language that combines first-order logic and relational calculus.

Imperative Programming:

- Describes *how* a program does something step by step.
- Sequential statements change a program's state.

Declarative Programming:

- Describes *what* a program should accomplish.
- There is no state changes or mutation.

The Alloy Language

Declarative Programming

- There is no state changes or mutation.
- *So, how do we represent dynamic behavior?*

```
x = 0
x' = x+1
```

The Alloy Language

Declarative Programming

- There is no state changes or mutation.
- *So, how do we represent dynamic behavior?*

```
x = 0
x' = x+1
```

actually:

```
x = 0
x' = x.add[1]
```

The Alloy Language

In Alloy, everything is a set! Common operators are reserved for set operations:

- $+$: set union
- $-$: set difference
- $=$: set equality
- $\&$: set intersection
- in : subset

The Alloy Language

- In alloy we use *signature* declarations to introduce `concepts` (sets of atoms).

```
sig Value {}

sig Key {
    map: Value
}
```

The Alloy Language

- In alloy we use *signature* declarations to introduce `concepts` (sets of atoms).

```
sig Value {}

sig Key {
    map: Value
}
```

- Signature fields, e.g., `map`, resemble OOP class members.

```
k.map // returns the value instance k maps to.
```

The Alloy Language

- In alloy we use *signature* declarations to introduce `concepts` (sets of atoms).

```
sig Value {}

sig Key {
    map: Value
}
```

- Signature fields, e.g., `map`, resemble OOP class members.

```
k.map // returns the value instance k maps to.
```

- `map` is a relation:

```
map : Key -> Value
```

The Alloy Language

In Alloy, everything, including sets, is a relation! Relational operators:

$P \to Q$	: arrow product	$\sim$	: transpose
$P.Q$	: dot product	$<:$	: domain restriction
$\hat{}p$	: transitive closure	$:>$	: range restriction
$*p$	: reflexive t.c.	$++$	: override

The Alloy Language

- Arrow product:

  $P \to Q$: every combination of tuples (p, q) for $p \in P$ and $q \in Q$.

The Alloy Language

- Arrow product:

  $P \to Q$: every combination of tuples (p, q) for $p \in P$ and $q \in Q$.

- Relation multiplicities:

  $P\ m \to n\ Q$: where m and n are multiplicity keywords: `one`, `lone`, `some`.

The Alloy Language

- Arrow product:

  $P \to Q$: every combination of tuples (p, q) for $p \in P$ and $q \in Q$.

- Relation multiplicities:

  $P\ m \to n\ Q$: where m and n are multiplicity keywords: `one`, `lone`, `some`.

- Example:

  $P \to$ `lone` Q : each member of P maps to zero or one member of Q.

The Alloy Language

- Arrow product:

  $P \to Q$: every combination of tuples (p, q) for $p \in P$ and $q \in Q$.

- Relation multiplicities:

  $P\ m \to n\ Q$: where m and n are multiplicity keywords: `one`, `lone`, `some`.

- Example:

  $P \to$ `lone` Q : each member of P maps to zero or one member of Q.

- Multi-arity relations:

  $P \to Q \to R$

Modeling ParamGen in Alloy

and verifying its correctness

Modeling ParamGen in Alloy

How to model a system in Alloy in an agile and incremental development style:

1. Specify the structure (i.e., the signatures)
2. Specify dynamic behavior.
3. Inspect instances.
4. Eliminate design flaws (e.g., by adding more constraints).
5. Analyze rigorously.

Modeling ParamGen in Alloy / Structure

Main ParamGen Structure: **Template**

- Stored in nested Python Dictionaries.

$$\underbrace{\{\text{Label, Guard}\}}_{\text{key}} \ : \ \underbrace{\{\text{Value, Dict}\}}_{\text{value}}$$

Modeling ParamGen in Alloy / Structure

```
sig Dict {
    contents: set Key
}

abstract sig Key {
    var map : lone Dict+Value
}

sig Value {}
```

Modeling ParamGen in Alloy / Structure

```
sig Dict {
    contents: set Key
}

abstract sig Key {
    var map : lone Dict+Value
}

sig Value {}
```

```
sig Label extends Key {}
sig Guard extends Key {}
sig Root extends Key {}
    {no this.~contents}
```

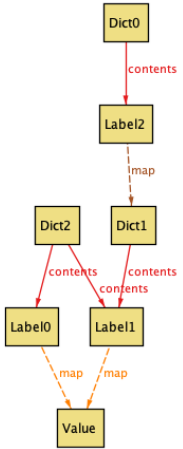
Modeling ParamGen in Alloy / Structure

```
sig Dict {
    contents: set Key
}

abstract sig Key {
    var map : lone Dict+Value
}

sig Value {}
```

```
sig Label extends Key {}
sig Guard extends Key {}
sig Root extends Key {}
    {no this.~contents}
```



Modeling ParamGen in Alloy / Constraints

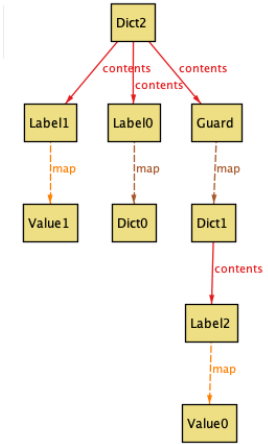
```
// each item can have at most one key
all i: Dict+Value |
    lone i.~map

// each key can be owned by one dict
all k: Key |
    lone k.~contents
```

Modeling ParamGen in Alloy / Constraints

```
// each item can have at most one key
all i: Dict+Value |
    lone i.~map

// each key can be owned by one dict
all k: Key |
    lone k.~contents
```



Modeling ParamGen in Alloy / Constraints

```
// each item can have at most one key
all i: Dict+Value |
    lone i.~map

// each key can be owned by one dict
all k: Key |
    lone k.~contents
```

```
// if a dict key is a guard,
// all dict keys must be so.
all d: Dict |
    {some Guard & d.contents implies
        d.contents in Guard}
```

Modeling ParamGen in Alloy / Constraints

```
pred invariants {
    // each item can have at most one key
    all i: Dict+Value |
        lone i.~map

    // each key can be contained by only one dict
    all k: Key |
        lone k.~contents

    // if a dict key is a guard, all dict keys must be so.
    all d: Dict |
        {some Guard & d.contents implies d.contents in Guard}

    // map.*contents relation is acyclic
    no iden & ^(map.*contents)

    // all values must be preceded by a label
    all v: Value |
        some v.^(~map.*~contents) & Label
}
```

Modeling ParamGen in Alloy / Dynamics

```
pred reduce[data: Dict+Value]{

  data in Dict implies {

    // (1) Expand vars in keys
    expand_vars[data.contents]

    // (2) Evaluate guards
    is_guarded_dict[data] implies
      impose_guards[data]

    // (3) Call reduce recursively
    else
      all key : data.contents |
        key.map' = key.map and
          reduce[key.map']
  }
  else
    // (4) Expand vars
    expand_vars[data]
}
```

```
def reduce(data):

  if isinstance(data, dict):

    # (1) Expand vars in keys
    data = expand_key_vars(data)

    # (2) Evaulate guards (if applicable)
    if is_guarded_dict(data):
      data = reduce(impose_guards(data))

    # (3) Call reduce recursively
    else:
      for key in data:
        data[key] = reduce(data[key])

  else:
    # (4) expand vars, apply formulas
    data = expand_vars(data)

  return data
```

Modeling ParamGen in Alloy / Dynamics

```
pred reduce[data: Dict+Value]{

  data in Dict implies {

    // (1) Expand vars in keys
    expand_vars[data.contents]

    // (2) Evaluate guards
    is_guarded_dict[data] implies
      impose_guards[data]

    // (3) Call reduce recursively
    else
      all key : data.contents |
        key.map' = key.map and
          reduce[key.map']
  }
  else
    // (4) Expand vars
    expand_vars[data]
}
```

```
def reduce(data):

  if isinstance(data, dict):

    # (1) Expand vars in keys
    data = expand_key_vars(data)

    # (2) Evaulate guards (if applicable)
    if is_guarded_dict(data):
      data = reduce(impose_guards(data))

    # (3) Call reduce recursively
    else:
      for key in data:
        data[key] = reduce(data[key])

  else:
    # (4) expand vars, apply formulas
    data = expand_vars(data)

  return data
```

Modeling ParamGen in Alloy / Dynamics

```
// Nondeterministically select one of the
// guards and drop all other branches
pred impose_guards[d: Dict]{
  let pkey = d.~map {
    some g: d.contents {
      pkey.map' = g.map and g.map'=none
      (d.contents-g).map' =
        (d.contents-g).map
      reduce[pkey.map']
    }
  }
}
```

```python
def impose_guards(self, data_dict):
    def _eval_guard(guard):
        """returns true if a guard evaluates to true."""
        assert isinstance(
            guard, str
        ), "Expression passed to _eval_guard must be string."

        if has_unexpanded_var(guard):
            raise RuntimeError("...")
        guard_evaluated = eval_formula(guard)
        assert isinstance(guard_evaluated, bool)
        return guard_evaluated

    if not ParamGen.is_guarded_dict(data_dict):
        return data_dict

    guards_eval_true = []  # list of guards that evaluate to true.
    for guard in data_dict:
        if guard == "else" or _eval_guard(str(guard)) is True:
            guards_eval_true.append(guard)

    if len(guards_eval_true) > 1 and "else" in guards_eval_true:
        guards_eval_true.remove("else")
    elif len(guards_eval_true) == 0:
        return None

    if self._match == "first":
        return data_dict[guards_eval_true[0]]
    if self._match == "last":
        return data_dict[guards_eval_true[-1]]
    raise RuntimeError("Unknown match option.")
```

Modeling ParamGen in Alloy / Dynamics

```
let expand_vars[expr]{
    no expr & varsToExpand'
}
```

```python
def expand_vars(expr, expand_func):

    if expand_func is None:
        return expr

    assert isinstance(
        expr, str
    ), "Expression passed to expand_vars must be string."

    expandable_vars = re.findall(r"(\$\w+|\${\w+\})", expr)
    for word in expandable_vars:
        ws = word.strip().\
            replace("$", "").replace("{", "").replace("}", "")
        word_expanded = expand_func(ws)
        assert (
            word_expanded is not None
        ), "Cannot determine the value of {}.".format(word)

        # enclose with quotes if expanded var is a string...
        if isinstance(word_expanded, str) and word[1] != "{":
            word_expanded = '"' + word_expanded + '"'
        else:
            word_expanded = str(word_expanded)

        expr = re.sub(
            r"(\$\b" + ws + r"\b|\$\{" + ws + r"\})",
            word_expanded, expr,
        )

    return expr
```

Modeling ParamGen Dynamics in Alloy / Abstract Representation

- Abstract representation of lower-level details allows us to focus on high-level software and algorithm design aspects.
  - Quick prototyping and early detection of design flaws!
  - Incremental modeling!

Modeling ParamGen Dynamics in Alloy / Abstract Representation

"The hard part of building software is the specification, design, and testing of conceptual construct, not the labor of representing it and testing the fidelity of the representation."

— FP Brooks. "No silver bullet". (1987)

Modeling ParamGen Dynamics in Alloy / Abstract Representation

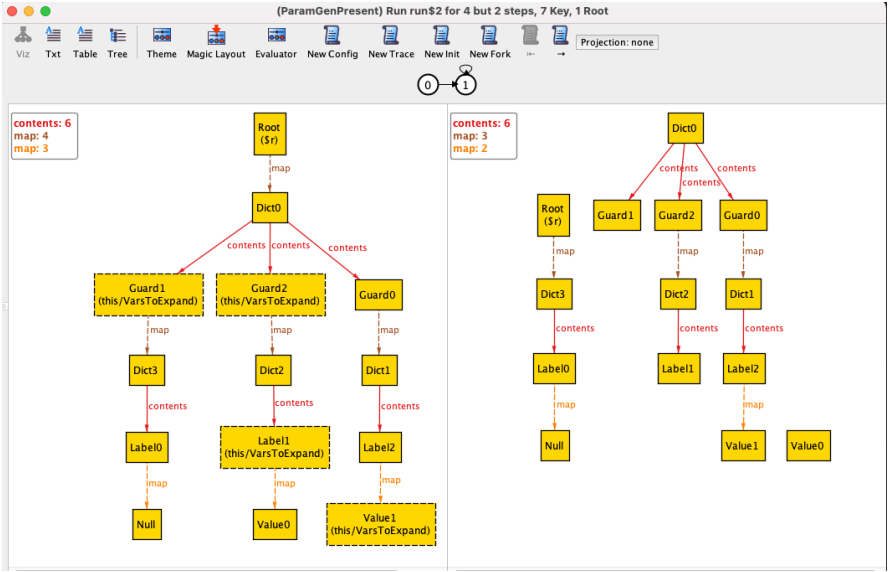
"The hard part of building software is the specification, design, and testing of conceptual construct, not the labor of representing it and testing the fidelity of the representation."

— FP Brooks. "No silver bullet". (1987)

"What matters is the fundamental structure of the design. If you get it wrong, there is no amount of bug fixing and refactoring that will produce a reliable, maintainable, and usable system."

— D Jackson. "The essence of software". (2021)

Back to ParamGen Dynamics...



Bounded Model Checking

```
reduce[r] implies {

    invariants and // All invariants still remain true
    r.map' in Dict // The result of postcondition should be a dict

    all ai:  r.*(map'.*contents) { // check active columns
        // all labels map to some Values or Dictionaries
        {ai in Label implies ai.map in Value+Null+Dict}

        // no active guard remains
        ai not in Guard

        // all keys should lead to a value
        {ai in Key implies some ai.^(map.*contents) & Value}

        // all values should have a label (varname)
        {ai in Value implies some ai.^(~map.*~contents) & Label}

        // no remaining vars to expand
        ai not in VarsToExpand'}
}
```

Bounded Model Checking

```
Executing "Check postconditions for 8 but 2 steps"
  Solver=glucose 4.1(jni) Steps=1..2 Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
  1..2 steps. 144942 vars. 657 primary vars. 338798 clauses. 31625ms.
  No counterexample found. Assertion may be valid. 97990ms.
```

Specifying a system helps us understand it.

"It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it."

— L Lamport. "Specifying Systems". (2002)

"Code is a poor medium for exploring abstractions."

— D Jackson. "Software Abstractions". (2006)

Specifying Scientific Software

- Specifying a system beforehand is not always possible in computational science.
  - Sometimes, we are handed over pre-developed code for integration.
  - Sometimes, complexity is underestimated and/or targets change.
  - Often, we work with legacy code.

Specifying Scientific Software

- Specifying a system beforehand is not always possible in computational science.
  - Sometimes, we are handed over pre-developed code for integration.
  - Sometimes, complexity is underestimated and/or targets change.
  - Often, we work with legacy code.
- Specification and modeling may be useful at later stages of development as well.
  - Modeling ParamGen in Alloy helped me better understand it (even though I was the original developer).

Specifying Scientific Software

- Specifying a system beforehand is not always possible in computational science.
  - Sometimes, we are handed over pre-developed code for integration.
  - Sometimes, complexity is underestimated and/or targets change.
  - Often, we work with legacy code.
- Specification and modeling may be useful at later stages of development as well.
  - Modeling ParamGen in Alloy helped me better understand it (even though I was the original developer).
- After developing the Alloy model of ParamGen:
  - I refactored the **reduce()** method and halved the LOC.
  - I have identified flaws in template handling and sanity checks.

Summary

Formal specification and software modeling can help us develop:

- Reliable software.
  - Testing is crucial, but incomplete.
  - Formal methods can be complementary to testing.
- Maintainable and efficient software.
  - Scientists and engineers are accustomed to working with models anyway.
  - With automatic, push-button analysis, one can focus on modeling and design aspects.
  - Design aspects matter. More so than lower level implementation details.

Announcement

November 9-10, 2023

Workshop on Correctness and Reproducibility for Climate and Weather Software

website: ncar.github.io/correctness-workshop/

Scope: Testing, Statistical Approaches, Formal Methods, Verification and Validation...

Venue: NCAR Mesa Lab & Virtual

Thanks!

altuntas@ucar.edu