Verifying ParamGen: A Case Study in Scientific Software Abstraction and Modeling

Alper Altuntas
altuntas@ucar.edu
National Center for Atmospheric Research
Boulder, CO, USA

John Baugh
jwb@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Jesse Nusbaumer
nusbaume@ucar.edu
National Center for Atmospheric Research
Boulder, CO, USA

Abstract

We introduce ParamGen, an infrastructure library for climate models to generate input files that specify physics, parameterizations, and other behavior. ParamGen supports arbitrary Python expressions for specifying a default set of runtime parameters and values, thereby providing a high level of expressiveness and flexibility. Initially developed for the MOM6 ocean component in Community Earth System Model (CESM), ParamGen is now used by several additional CESM components. Therefore, it is of high importance that it operates correctly, i.e., absent any undesired and unexpected behavior. To that end, we develop an abstract verification model of ParamGen in Alloy, a software modeling and analysis tool with a declarative language that combines first-order logic and relational calculus. We evaluate the correctness of ParamGen via Alloy and discuss how abstract models and formal verification can help quickly frame questions and get answers regarding the structure and behavior of our scientific computing applications. We also describe our experience in coming to a cleaner and well-formed software design and abstractions as a result of the modeling exercise we present in this study.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: Alloy, CESM, Formal Specification, Scientific Computing

ACM Reference Format:
Alper Altuntas, John Baugh, and Jesse Nusbaumer. 2023. Verifying ParamGen: A Case Study in Scientific Software Abstraction and Modeling. In *Proceedings of 2023 Improving Scientific Software Conference (ISS'23).* NCAR, Boulder, CO, USA, 9 pages. https://doi.org//10.5065/j0e4-ss70

1 Introduction

Earth system models are typically configured with multiple runtime input parameter files to specify model physics, parameterizations, numerics, and other behavior. These files may contain tens to hundreds of parameters with intra- and inter-component dependencies and constraints. Therefore, modeling frameworks, such as that of the Community Earth System Model (CESM), include infrastructure libraries that automatically generate default input parameter files corresponding to consistent and valid configurations. In this study, we introduce one such infrastructure library, called ParamGen. This library supports arbitrary Python expressions for specifying default runtime parameters and values, thereby providing model developers with a high level of expressiveness and flexibility.

We initially developed ParamGen as part of an effort to incorporate the MOM6 ocean model into CESM. MOM6 is developed and maintained by a consortium of ocean modelers with members from NOAA, NCAR, NASA, and other modeling centers and universities. While the core MOM6 codebase is shared by all members of the consortium, each modeling center couples MOM6 with a unique set of components by using distinct coupling infrastructures and superstructures. The motivation for the development of ParamGen, therefore, arose from a need to bring together the well-established conventions and workflows of MOM6 and those of CESM. For instance, MOM6 expects at least three input parameter files, each having a vastly different syntax and corresponding to different aspects of the model such as parameterizations, logistics, and diagnostics. CESM, on the other hand, assigns a single input parameter file for each component where users may specify configuration customizations. Additionally, CESM allows users to make changes in various intra- and inter-component settings via the execution of a Python script called `xmlchange`. ParamGen ensures that any changes made by the user through CESM mechanisms are appropriately and coherently propagated to the MOM6 input files. For instance, when users specify the ocean coupling interval via the `xmlchange` command, ParamGen ensures that the internal MOM6 timesteps are compatible with CESM's ocean coupling interval: In the absence of such compatibility, the simulation may lead to instabilities or invalid model results.

ParamGen has recently been incorporated in CESM's case control system called the Common Infrastructure for Modeling the Earth Systems (CIME) [11]. Its incorporation into CIME has facilitated its adoption by several additional components such as the Community Atmosphere Model (CAM). Before the adoption of ParamGen, CAM developers maintained multiple markup files and an ad hoc Perl script to

ISS'23, April 2023, Boulder, CO, USA 2023.
https://doi.org/10.5065/j0e4-ss70

specify and evaluate parameter values that depend on model grids and other physics packages. Adoption of ParamGen has eliminated the need to maintain multiple markup files and an ad hoc script, which consequently improved the maintainability and ease of use.

Given its critical role in ensuring the consistency and validity of model parameters, it is of high importance that ParamGen functions reliably, i.e., that it is bug-free and operates as expected at all times. To ensure the reliability of ParamGen, CESM developers routinely carry out unit and integration tests. However, testing is incomplete:

> "Testing can be used to show the presence of bugs, but never to show their absence [6]."

In this study, we describe a *lightweight formal methods* approach to gain further confidence. Specifically, we present a formal specification of ParamGen in Alloy, a software modeling and analysis tool with a declarative language that combines first-order logic and relational calculus [8]. Using the automatic analysis feature of Alloy, we check for violations of a number of safety conditions that we specify and confirm that no counterexample exists. In addition to ensuring further reliability and confidence, we discuss several additional benefits of formal specification, such as coming to a cleaner software architecture and well-formed abstractions and concepts.

In the next section, we describe the ParamGen library in detail. We then provide brief background information on Alloy, which is followed by the description of our Alloy model of ParamGen. We demonstrate how Alloy Analyzer can quickly check safety conditions and ensure the correctness of the structure and behavior of our scientific computing applications. As an added benefit, we discuss how formal specification improves our understanding of the software systems we build, through which we can end up with better-formed abstractions and clearer code implementations.

2 ParamGen

ParamGen is a lightweight, generic Python module for generating runtime input parameter files for earth system modeling applications. The ParamGen module supports arbitrary Python expressions for the specification of model input parameters. This provides a high level of flexibility and genericity.

ParamGen infers parameter values from templates, i.e., generic and comprehensive specifications of parameters and values. Templates are typically specified in a markup format and put together and maintained by the model developers. ParamGen is agnostic of any implementation-specific details of modeling frameworks, components, or input/output formats. Yet several widely used markup formats are readily available in ParamGen as supported template formats. These are xml, yaml, and json formats. The only out-of-the-box output format, on the other hand, is the Fortran namelist format. New input and output formats can easily be introduced by application developers via class inheritance.

2.1 ParamGen Templates

A complete ParamGen template must contain all the necessary information to generate a model input parameter file for all cases. In the simplest case, a template entry consists of a key-value pair:

```
NIHALO: 2
```

In the above template entry, `NIHALO` corresponds to a model parameter whose value is to be set to `2` in all cases. If instead, the value of a particular parameter is dependent on certain criteria, we can specify so by listing value alternatives and by preceding each value alternative by a *guard*, i.e., a logical statement that determines whether the following value is a valid choice. For example:

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

Depending on which preceding guard evaluates to true, the `NIGLOBAL` parameter above gets set to either `320` or `540`. The guards for the `NIGLOBAL` variable are mutually exclusive. But in other parameter specifications where multiple guards may evaluate to true at the same time, the default (but configurable) ParamGen behavior is to pick the last valid value.

While the guards above are equality checks, a ParamGen guard may be any arbitrary Python expression that evaluates to true or false. The `OCN_GRID` variable appearing in both guards above corresponds to the ocean model grid, a high-level model setting predetermined by the user while creating an experiment, i.e., an instance of a model simulation. `OCN_GRID` is an example of an *expandable variable*. In ParamGen, expandable variables work similarly to how they work in shell scripting languages. When ParamGen reads in and interprets a template entry, it replaces the occurrences of expandable variables with their actual values.

Guards are useful when the set of *guard: value* alternatives is small. If, however, the set of alternatives is large or infinite, we may instead choose to specify all value alternatives via a generic, single Python formula:

```
DT_TERM:
  = (($PERIOD =="decade") * 86400 * 3650 +
     ($PERIOD =="year") * 86400 * 365 +
     ($PERIOD =="day") * 86400 +
     ($PERIOD =="hour") * 3600 )/ $OCN_NCPL
```

The DT_THERM parameter above corresponds to the MOM6 thermodynamic timestep whose value is specified as a Python expression involving the coupling period PERIOD and ocean coupling interval OCN_NCPL, both of which are high-level settings predetermined by the user before ParamGen comes into play.

Internally, ParamGen templates are stored in nested Python dictionaries where keys may correspond to labels (variable names, namelist groups, property names, etc.) or guards. The distinction between labels and guards is that guards are arbitrary Python expressions evaluating to true or false while labels are either string literals or arbitrary Python expressions that evaluate to strings.

2.2 ParamGen Schema

The below list summarizes some of ParamGen's features that make it quite flexible when it comes to specifying a template.

- Expandable variables and arbitrary Python expressions can appear anywhere in a template: in labels, guards, or value specifications.
- There can be multiple labels along a branch, e.g., to specify namelist groups. parameter names, and parameter properties.
- There can be multiple (nested) guards along a branch.
- There is no restriction on the ordering of guards and labels: Guards can appear before, after, or in between labels and vice versa.

Nevertheless, there are a couple of schema rules that must be adhered to by all template specifications:

1. There must be at least one label along each template branch.
2. If a key is a guard, then all of its siblings (i.e., keys of the same dictionary) must also be guards.

As an example, the below template entry violates the second schema rule since `description` and `$OCN_GRID == "g16"` (a label and a guard, respectively) are keys of the same dictionary:

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

This violation may be eliminated by, for instance, adding a `value` label that precedes the guard specification:

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

Here, the intention is to eliminate any semantic ambiguity that may result from specifying labels and guards at the same level.

2.3 The `reduce()` method

The main ParamGen operation is the `reduce()` method that reads in and interprets a template specification and generates the input parameter file depending on some high-level model settings predetermined by the user. The `reduce()` method operates by recursively traversing all the keys and values of the template and applying in-place modifications such as expanding variables, evaluating arbitrary Python expressions, and imposing guards by dropping template branches including guards that evaluate to false.

As a recursive operation with in-place data modifications, the `reduce()` method brings about substantial algorithmic complexity. This complicates software assurance efforts. Another factor that contributes to this complexity is that ParamGen allows for great flexibility in specifying templates, thereby requiring many different layouts and topologies to be accounted for. These factors motivate our exploration of *lightweight formal methods* and Alloy as a complementary means to testing for further confidence.

3 Alloy

Alloy is a software modeling and analysis tool with a declarative language that combines first-order logic and relational calculus [8]. The Alloy language is simple, precise yet powerful and quite elegant. As a high-level modeling language, Alloy allows modelers to specify software systems at an abstract level, i.e., at the software and algorithm design level, and above the implementation-specific details.

Alloy is a declarative language, so it doesn't allow state changes. However, dynamic behavior may be emulated by using the prime (′) operator which is used to denote the value of a variable at a (virtual) next state.

In Alloy, everything is a set. Therefore, common operators such as + and – are reserved for set operations. For arithmetic addition and subtraction, for instance, the modelers are to use functions add and sub, respectively. For example, the below Alloy statement increments the value of an arbitrary variable x by one. Note the usage of the prime (′) operator to refer to the value of x at the next state.

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

In Alloy, new sets may be introduced via signature declarations. Similar to classes in object-oriented languages, signatures can have members, or fields in Alloy nomenclature. Below is an example declaration of a signature A, that has a field called x, which points to an instance of another signature B:

```
sig A {
    x: B
}
```

The x field of an arbitrary instance a of A may be accessed via the dot (.) operator:

```
a.x
```

Fields may be thought of as relations. In the above example, x is a relation from A to B. As a language based on relational calculus, Alloy provides a number of relational operators, one of which is the arrow product that may be used to refer to a relation from a set P to set Q:

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

Relational multiplicity keywords may be used to restrict the multiplicities of a relation. For instance:

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

Some of the remaining Alloy features and operations will be briefly described in the next section as we encounter them in our Alloy model of ParamGen. For detailed instructions on the Alloy language and analyzer, users are referred to an online tutorial [5].

4 Modeling ParamGen in Alloy

For the development of the Alloy model of ParamGen, we adhere to a customary workflow. As summarized in Figure 1, the main steps of the workflow consist of specifying the two major aspects of any software system: structure and dynamics. After these aspects are specified, users may instruct Alloy to generate arbitrary instances and transitions for visual analysis. Finally, an automated and rigorous analysis is carried out by Alloy to check for any assertion violations within a given bound.

4.1 Structure

As a first step, we specify the main ParamGen structure in Alloy, that is the *template* data structure. In the original ParamGen source code, templates are implemented via nested Python dictionaries. As a simple and abstract language, Alloy doesn't intrinsically include compound data structures like dictionaries or hash tables. Therefore, we begin our modeling effort by first specifying an abstract representation of nested Python dictionaries:

```
sig Dict {
    contents: set Key
}

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

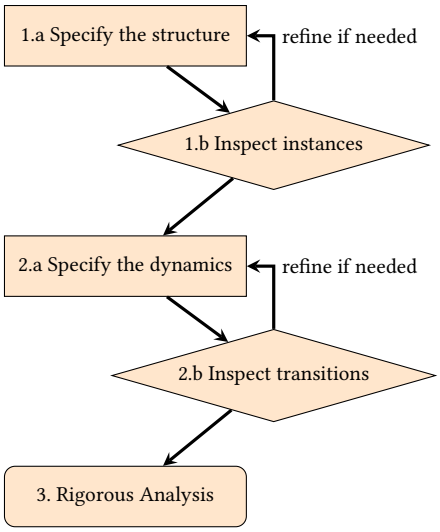

Figure 1. A customary workflow for modeling a software system in Alloy. Steps 1.a and 2.a correspond to user input in the form of model specification. In steps 1.b and 2.b, arbitrary model instances generated by Alloy are visually inspected by the users to identify and address any apparent flaws. In the final step, the Alloy analyzer automatically checks for all possible states and transitions within a given bound.

```
sig Value {}
```

The Dict signature above models Python dictionaries. Its contents field is a set of Keys, which is subsequently defined as a signature with a *mutable*[1] field called map. Dictionary keys can map to either values or other dictionaries (in the case of nested dictionaries). Thus, the map field points to (zero or one member of) set union of Dict and Value signatures, the latter of which is specified to have no fields and corresponds to an abstract representation of parameter values.

Notice that the Key signature definition is preceded by the abstract keyword. Abstract signatures are similar to abstract classes in object-oriented languages: They are used as base signatures for derived signatures (or, extensions in Alloy nomenclature). Below are the three extensions of the Key signature:

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

All three extensions above have empty signature bodies, so they don't have any fields other than the map field that they inherit from the Key signature. The Label signature corresponds to dictionary keys that are of type string, while

[1]The var keyword designates fields and signatures as mutable entities.

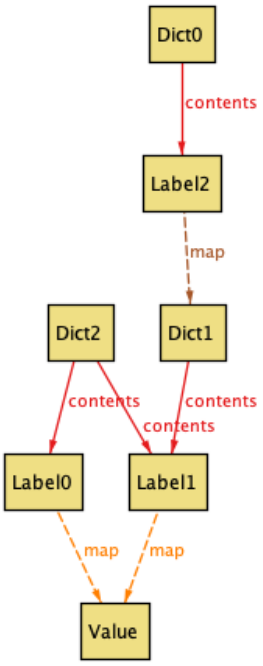

Figure 2. An arbitrary Alloy model instance satisfying the initial ParamGen structure specification.

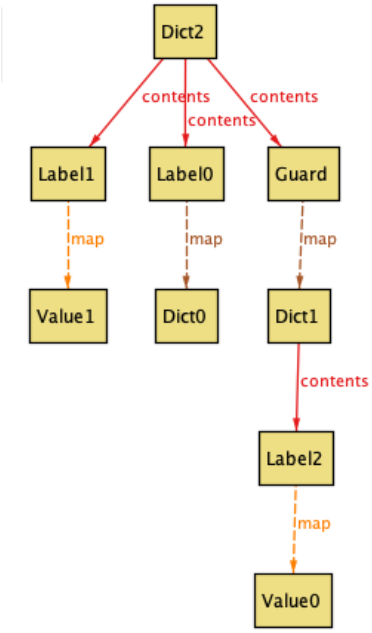

Figure 3. An arbitrary Alloy model instance satisfying the initial ParamGen structure specification and the first set of assertions.

the `Guard` signature corresponds to keys that are logical expressions. The `Root` key is used as a modeling shortcut to refer to the outermost dictionaries in nested dictionaries. Notice that a fact[2] statement is attached to the `Root` signature definition. This fact statement makes use of an existential quantifier `no` (empty set) and relational transpose operator (~). Simply, it states that the set of dictionaries that contain root key instances is an empty one.

Having defined all the conceptual entities of the ParamGen structure, we can now instruct the Alloy analyzer to generate arbitrary instances that satisfy our structure specification. One such model instance is shown in Figure 2, where two issues stand out: `Label1` is simultaneously contained in two different dictionary instances, `Dict1` and `Dict2`. Similarly, the `Value` instance is mapped to by two distinct label instances, `Label0` and `Label1`.

To eliminate these flaws, we specify the following assertions:

```
// 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
```

[2]Facts are model constraints that are specified to hold at all times.

The first assertion above states that for each instance `i` of the set union of dictionaries and values, there exists at most one (`lone`) key that maps to `i`. Here, we make use of the relational transpose operator ~ to invert the `map` relation and, so, to refer to keys that map to `i`. The second assertion above states that for each `Key` instance `k`, there exists at most one dictionary that contains `k`. In the latter assertion, we again use the relational transpose operator ~ to invert the `contents` relation and to refer to dictionaries that contain `k`.

Having incorporated these facts, we instruct Alloy to generate new model instances and visually confirm that none of the instances exhibit these structural flaws. However, one of the resulting instances, shown in Figure 3, exhibits another issue: Contrary to the second schema rule defined in Section 2.2, the outermost dictionary instance `Dict2` contains both `Label` and `Guard` instances.

We eliminate this schema rule violation by adding another assertion as a fact. This new assertion states that for each dictionary instance `d`, all of the contents (`d.contents`) must be guards if at least one (`some`) of its keys is a guard:

```
// 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}
```

After going through this iterative process of refining the model and visually inspecting arbitrary instances, we end

up with a full set of assertions shown below. Note that in the remainder of this paper, we skip the detailed explanation of Alloy expressions as well as the keywords and operators used in them. Instead, we briefly describe each statement via preceding comment lines.

```
pred invariants {

    // 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}

    // 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
}
```

We specify these assertions within a predicate[3] called `invariants`, so as to reuse them as a fact (precondition) and as a postcondition during the model checking phase.

4.2 Dynamics

The main operation in ParamGen is the `reduce()` method that traverses all of the template branches recursively and applies in-place modifications, e.g., expanding variables, evaluating arbitrary Python expressions, and imposing guards, to generate the final set of *parameter:value* pairs to go into the model input file.

Rather than providing an in-depth description of the Alloy specification and the Python implementation of the `reduce()` method, we present them side-by-side in Listing 1, and make the following observations: Both Alloy and Python versions have highly readable and succinct syntaxes albeit quite distinct since each language corresponds to a different programming paradigm and serves different purposes. Syntaxes aside, both versions of the `reduce()` method look quite similar due to a couple of reasons: Firstly, the `reduce()` method corresponds to higher level code where algorithm design aspects are expressed, and where implementation-specific details are hidden beneath lower-level function calls like `impose_guards()` and `expand_vars()`.

Another reason that the Alloy and Python versions are alike is that the Python version shown in Listing 1 is the result of a refactorization that followed the Alloy modeling exercise. This refactorization made the Python version much more concise and clear compared to the original implementation which had more than twice the number of lines of code. Coming to a much cleaner software and architecture is in fact an established benefit of formal specification and abstraction. A similar experience was described by the leader of a team that built a real-time operating system: A ten-fold reduction in code size was achieved after going through an exercise of formal specification [10].

While the Alloy and Python versions of the higher-level `reduce()` method are quite alike, the same cannot be said for the versions of lower-level operations. The Python version of the `impose_guards()` method, for instance, has 25 LOC (not shown) whereas the Alloy version is much more concise:

```
// 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']
    }
  }
}
```

Similarly, the Alloy version of `expand_vars()` method contains only a single expression, whereas the Python version (not shown) contains a variety of control structures, string manipulations, and regular expression operations.

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

The Alloy versions of these lower-level operations are merely abstract representations of actual Python implementations. Instead of describing what each operation should do in a step-by-step fashion, they just specify what should be accomplished at an abstract level. Such abstract representation of lower-level details allows us to focus on high-level software and algorithm design aspects. As a result, we can quickly develop prototypes and detect any design flaws at the early stages of development. This also allows us to adapt an incremental modeling style where we can incorporate more and more details as we develop and refine our design.

Before we conclude the specification of ParamGen dynamics, we provide an example of an arbitrary transition that

[3]Predicates may be thought of as boolean functions returning true or false depending on the evaluation of the statements that they encompass.

```
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) Evaluate guards
    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
```

Listing 1. Alloy and Python versions of ParamGen's `reduce()` method are shown on the left and right panels, respectively.

demonstrates the effect of executing the `reduce` method on an arbitrary template instance (Figure 4).

4.3 Bounded Model Checking

Visually inspecting a number of arbitrary model instances is quite useful in refining the early versions of an Alloy model. As the model evolves, however, a more rigorous analysis may be carried out by the Alloy analyzer in the form of *bounded model checking*. In this mode of analysis, the Alloy analyzer interprets the model specification and automatically checks for any violations of assertions by searching through all possible states and transitions within a given bound.

For our Alloy model of ParamGen, we identify and specify the safety conditions as follows: (each assertion is briefly described via adjacent comment lines.)

```
reduce[r] implies {

    invariants and // Invariants remain valid
    r.map' in Dict // The result is a Dict

    // ai: active items
    all ai:  r.*(map'.*contents) {

        // all labels map to some Value or Dict
        {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
        {ai in Value implies
            some ai.^(~map.*~contents) & Label}

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

Having specified all the necessary safety conditions of interest, we instruct Alloy to check for any violations for at most 8 instances of each top-level signature, i.e., `Dict`, `Key`, and `Value`. In about a couple of minutes, Alloy analyzer confirms that no counterexample exists within the given bound (Figure 5).

A small bound on the scope might sound insufficient in detecting real-world bugs. However, the *small scope hypothesis* states that "most bugs have small counterexamples" and so if an assertion is invalid, it can most likely be replicated within a small scope [8]. Thus, bounded model checking can provide further confidence when combined with traditional testing.

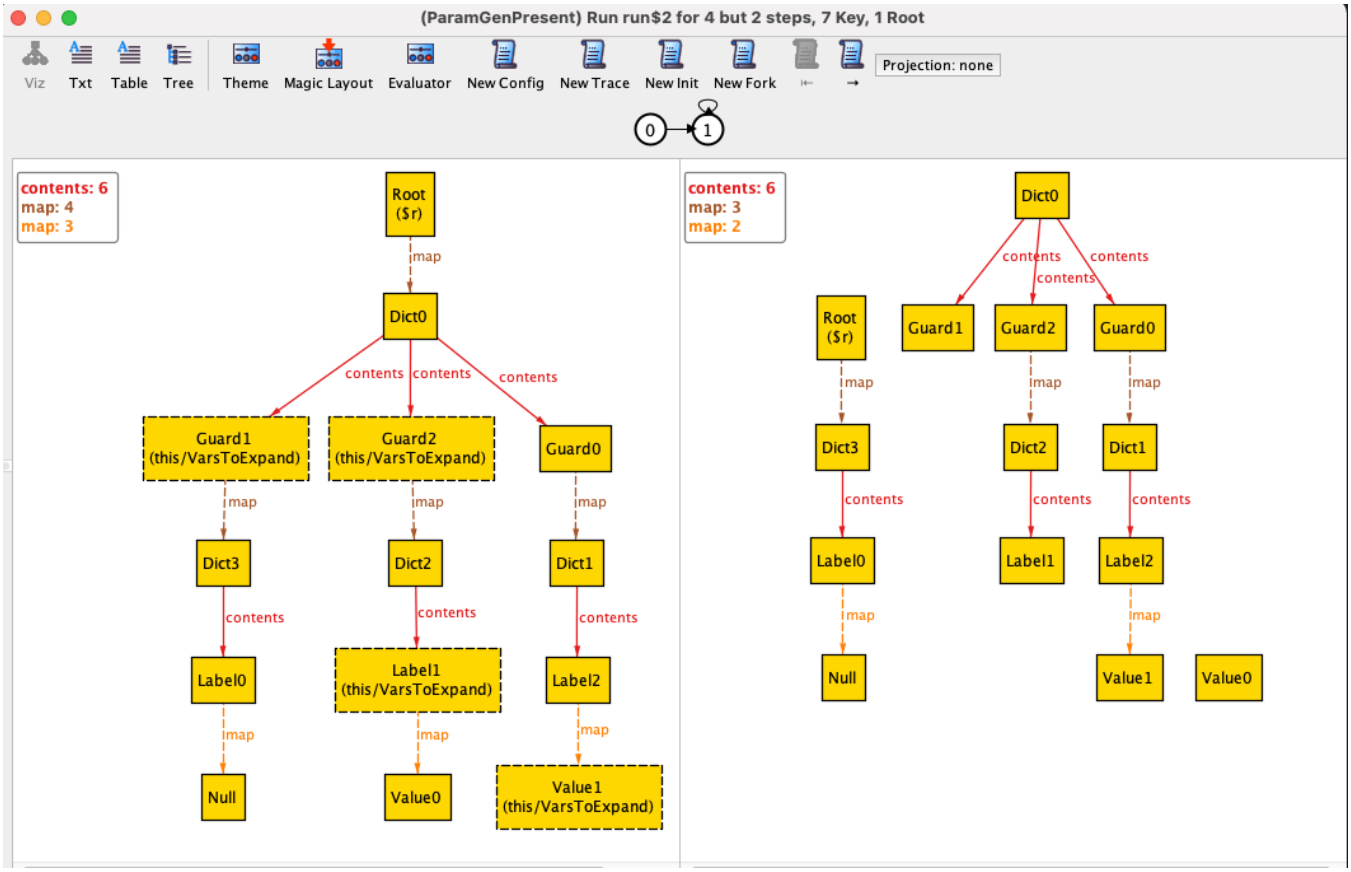

Figure 4. An arbitrary transition generated by Alloy. The tree beginning with the `Root` key on the left panel corresponds to an initial, arbitrary template instance, while the tree beginning with the `Root` key on the right panel corresponds to the final outcome of executing the `reduce()` method.

```
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.
```

Figure 5. Alloy confirms that no safety violation occurs within the given bound.

5 Discussion

As discussed in the previous section, a major benefit of formal methods, and of Alloy, in particular, is further confidence in the reliability of software. Another major benefit is that formal specification helps us better understand the systems that we develop.

> "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 [9]."

Although specifying a system beforehand is the ideal software development approach, it may not always be feasible or possible in the realm of scientific computing where short-term concerns about performance, resources, and project timelines dominate and legacy codebases are widely encountered [3]. Yet, we believe that formal specification and modeling may be useful at later stages of scientific software development as well.

In fact, this study is one such attempt at utilizing formal methods: We developed the Alloy model after we developed the Python implementation and made it operational. Even so, we have benefited significantly from this modeling exercise. Firstly, we have come up with cleaner and more maintainable software as a result of refactoring guided by the Alloy specification. Additionally, we have identified flaws in template handling and extended sanity checks in the actual implementation to prevent unexpected or unaccounted-for template

formations. For instance, the first schema rule defined in Section 2.2 was identified during the development and analysis of the Alloy model of ParamGen. Subsequently, it has been incorporated into the Python implementation as a safety condition.

6 Conclusions

We have introduced a highly flexible Python module called ParamGen for auto-generating runtime input parameter files within earth system modeling frameworks. Subsequently, we have developed and analyzed a formal specification of ParamGen in Alloy to reason about its behavior and reliability. In doing so, we have observed and argued for several benefits of formal specification that can be summarized in two broad themes:

Reliability. Testing is crucial in developing reliable scientific software, but it is not always sufficient. Due to their rigorous nature and coverage that cannot be matched by any testing suite, formal methods may be complementary to testing wherever feasible.

Maintainability. Exploring abstractions that form the basis of software design via formal specification lead to more maintainable codebases. This is evidenced by our experience in refactoring ParamGen's `reduce()` method and coming up with a cleaner and more concise version after going through the exercise of modeling it in Alloy.

As far as we are aware, ours is the first study to formally specify and analyze a parameter generator used within the context of climate modeling. However, we have previously used formal methods to specify and analyze other aspects of climate models and scientific computing applications. For example, we have previously used KeYmaera X, a popular formal methods tool from the field of cyber-physical systems, to verify the correctness of discrete updates that appear in the K-profile parameterization, a vertical ocean mixing scheme used in MOM6 and several other ocean models [2]. Similarly, we used Alloy to analyze a discrete wetting and drying algorithm used in a hurricane storm surge model [3]. We also used Alloy to model and reason about sparse matrix computations, which are central to climate models and many other scientific computing applications [4, 7]. In another study, we used the SPIN model checker to specify the inherent concurrency and verify the absence of race conditions in a multi-instance ocean model [1].

A single approach or tool may not be able to meet every need [2]. However, as demonstrated by this and prior work, we believe that formal specification and verification can be supplemental and practical means of improving the reliability and maintainability of many aspects of earth system models, and, more broadly, of scientific computing applications.

References

[1] Alper Altuntas and John Baugh. 2017. Verifying concurrency in an adaptive ocean circulation model. In *Proceedings of the First International Workshop on Software Correctness for HPC Applications*. 1–7.

[2] Alper Altuntas and John Baugh. 2018. Hybrid theorem proving as a lightweight method for verifying numerical software. In *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 1–8.

[3] John Baugh and Alper Altuntas. 2018. Formal methods and finite element analysis of hurricane storm surge: A case study in software verification. *Science of Computer Programming* 158 (2018), 100–121.

[4] Juan Benavides, John Baugh, and Ganesh Gopalakrishnan. 2023. An HPC practitioner's workbench for formal refinement checking. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 64–72.

[5] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. 2021. Formal Software Design with Alloy 6. https://haslab.github.io/formal-software-design/index.html. Accessed: 2023-05-07.

[6] Edsger Wybe Dijkstra et al. 1970. Notes on structured programming.

[7] Tristan Dyer, Alper Altuntas, and John Baugh. 2019. Bounded verification of sparse matrix computations. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 36–43.

[8] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[9] Leslie Lamport. 2002. Specifying systems: the TLA+ language and tools for hardware and software engineers. (2002).

[10] Leslie Lamport et al. 2018. If you're not writing a program, don't use a programming language. *Bulletin of EATCS* 2, 125 (2018).

[11] CIME Repository. 2023. https://github.com/ESMCI/cime Accessed: 2023-06-16.