An HPC Practitioner's Workbench for Formal Refinement Checking

Juan Benavides[1], John Baugh[1(✉)], and Ganesh Gopalakrishnan[2]

[1] North Carolina State University, Raleigh, NC 27695, USA
{jdbenavi,jwb}@ncsu.edu
[2] University of Utah, Salt Lake City, UT 84112, USA
ganesh@cs.utah.edu

Abstract. HPC practitioners make use of techniques, such as parallelism and sparse data structures, that are difficult to reason about and debug. Here we explore the role of data refinement, a correct-by-construction approach, in verifying HPC applications via bounded model checking. We show how single program, multiple data (SPMD) parallelism can be modeled in Alloy, a declarative specification language, and describe common issues that arise when performing scope-complete refinement checks in this context.

Keywords: Formal methods · scientific computing · parallelism · Alloy

1 Introduction

To explain the points of view expressed in this paper, it helps to take the example of how an HPC expert practices their programming craft. The expert has the intended math in their head but then uses the medium of code not just to express intent but also to get the code running efficiently on the chosen computational medium, be it a CPU or a GPU accelerator. Such details end up being baked into the code, including sparse data structure designs and thread-to-array-slice mappings. Even though the final product is arrived at through a succession of *refinements*, it is seldom that these intermediate forms play a continued role in explaining the elaboration of the design. Doing so may in fact be considered counterproductive, since one would be forced to maintain even more code.

While this practice is standard and seemingly successful in some ways, one has to question whether the required apprenticeship is keeping a generation of (otherwise programming-language-aware) students from entering the area. Even absent this, the real price seems to be already getting paid by the experts when they suddenly realize that porting the code to a new platform requires hard-to-hire talent, with bugs crippling productivity [7]. Performance-portability mechanisms such as Kokkos [4] and RAJA [3] seem like a possible answer, but they are not widespread, and are too much to teach at an introductory level.

C. Mendis and L. Rauchwerger (Eds.): LCPC 2022, LNCS 13829, pp. 64–72, 2023.
https://doi.org/10.1007/978-3-031-31445-2_5

Harking back to the vision of early computer science pioneers, and taking inspiration from the creators of lightweight model-finding "thought calculators" such as Alloy [9], we present our experience-to-date using Alloy as the medium in which to capture refinements. We argue that doing so might encourage the practice of stating refinements in the tangible (and analyzable) medium of formal logic. The benefits of Alloy-style specifications are already evident in their ability to generate test cases—even for GPU memory models [10]—and perhaps they may one day help formally examine code ports at the design level.[1]

Scope and Organization. In what follows, we describe a lightweight modeling approach for reasoning about the structure and behavior of scientific software. We propose abstraction and refinement principles to manage sources of complexity, such as those introduced to meet performance goals, including sparse structure and parallelization. Elements of the approach include declarative models that are automatically checked with (Boolean satisfiability) SAT solvers, akin to the analysis approaches used in traditional engineering domains, so no theorem proving is required. The approach is bounded and therefore incomplete, but we appeal to the *small scope hypothesis*, which suggests that most real bugs have small counterexamples. For data refinement, we adopt a state-based style, which extends well to concurrency and parallelism—typically better than, say, an algebraic one.

We begin with related work, then introduce our refinement-checking approach and demonstrate it in the context of an HPC application, and follow up briefly with conclusions and future directions.

2 Previous Work

Formal methods is an extensive field that we do not intend to survey. Instead, we present a few examples of related work that are most relevant to the HPC community. These studies set a precedent for the framework we present, highlighting refinement, lightweight model-finding, and rich state in scientific computing.

Dyer et al. [5] explore the use of Alloy to model and reason about the structure and behavior of sparse matrices, which are central to scientific computing. Examples of sparse matrix-vector multiplication, transpose, and translation between ELLPACK and compressed sparse row (CSR) formats illustrate the approach. To model matrix computations in a declarative language like Alloy, a new idiom is presented for bounded iteration with incremental updates. The study considers the subset of refinement proof obligations that can be formalized as safety properties—and are thus easier to check—in Alloy.

Baugh and Altuntas [2] describe a large-scale hurricane storm surge model used in production and verification of an extension using Alloy. To explore implementation choices, abstractions are presented for relevant parts of the model, including the physical representation of land and seafloor surfaces as a finite element mesh, and an algorithm that allows for the propagation of overland flows. Useful conclusions are drawn about implementation choices and guarantees about the extension, in particular that it is equivalence preserving.

[1] It is well known that running unit tests is a poor way of unearthing conceptual flaws.

Martin [11] shows how a data refinement approach can be used to formally specify parallel programs using a Coarray Fortran (CAF) implementation of an iterative Jacobi routine. At an abstract level, a mathematical description of a step in the iteration is given, and at the concrete level, the corresponding operation is defined for parallel coarray images; an abstraction function relates the two levels. Since it focuses on specification, the roles of state-space invariants and other refinement proof obligations needed for verification are not addressed.

3 Approach

Data refinement is a correct-by-construction approach for the stepwise development of programs and models from a higher-level abstract specification to lower-level concrete ones [13]. The HPC field lends itself well to a data refinement approach as most programs begin with a mathematical specification—often as a theory report—that serves as a guide, to one extent or another, in the implementation of high performance code.

To ensure that a refinement step preserves correctness in a formal, machine-checkable manner, proof obligations must be met and discharged; their articulation and promotion begins with the work of Hoare [8] and thereafter proceeds along both relational and predicate transformer lines; de Roever [13] summarizes and contrasts a variety of modern approaches. While many of these offer some degree of tool support, Alloy's model-finding strengths for expressing rich state, combined with its push-button automation, make it an attractive alternative to those requiring theorem proving, especially for HPC practitioners.

Below we introduce refinement checking and some of the practical details of formalizing the checks in Alloy. Our goal beyond this short paper is to develop a general framework for carrying out data refinement checks in Alloy that will make the approach clear and appealing to practitioners. That includes characterizing a sufficient set of proof obligations for data refinement, showing how to encode them in Alloy, and demonstrating the approach on practical HPC problems.

4 Refinement Checking

The notion of refinement is relative. It makes use of an upper abstract level and a lower level concrete one:

> *Definition: Semantic Implementation Correctness* [13]. Given two programs, one called concrete and the other called abstract, the concrete program *implements* (or *refines*) the abstract program correctly whenever the use of the concrete program does not lead to an observation which is not also an observation of the abstract program.

Refinement as inclusion, above, is a global criterion. To be made practical, a local criterion with a finite number of verification conditions can be obtained by defining a *simulation* in terms of abstraction relations and commutativity diagrams.

In Fig. 1, the commutativity diagram shows the concrete (C) and abstract (A) states related by an abstraction relation α, together with concrete and abstract operations, op_c and op_a, respectively, that define transitions from the non-primed to primed states.

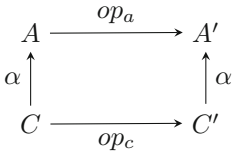

Fig. 1. Data Refinement

There are four different technical notions of simulation that correspond to ways in which commutativity can be defined in terms of the diagram [13]. When α is both total and functional, the four types of simulation coincide and some of the proof obligations simplify, including the condition for *correctness* of the concrete operation op_c:

$$\forall\, a, a' : A,\ c, c' : C \mid \alpha(c, a) \wedge op_c(c, c') \wedge \alpha(c', a') \Rightarrow op_a(a, a') \tag{1}$$

That is, starting from a concrete state in which the corresponding abstract precondition holds, the final concrete state must represent a possible abstract final state. Such a criterion implies inclusion, i.e., that programs using some concrete data type C have (only) observable behaviors of programs using a corresponding abstract data type A. Diagrams satisfying such properties are said to commute *weakly*, whereas strong commutativity would be expressed with material equivalence instead of implication in Eq. 1.

Summarizing the set of proof obligations for data refinement in a state-based formalism [13], we have the following conditions:

1. *Adequacy* – every abstract state must have a concrete counterpart.
2. *Correspondence of initial states* – every concrete initial state must represent an abstract initial state.
3. *Applicability of the concrete operation* – the precondition for the concrete operation should hold for any concrete state whose corresponding abstract state satisfies the abstract precondition.
4. *Correctness of the concrete operation* – as we describe above.

Not every condition applies in every situation, and in some cases a condition may require a special interpretation for the given context. For instance, an adequacy check for a refinement from abstract matrices to coarrays can be satisfied trivially in the one-processor case: a single coarray matrix, equivalent to the abstract one, "refines" it, but one might rather show adequacy for an n-processor case.

To draw sound conclusions from these, the structure of α is clearly important. If the correctness condition of Eq. 1 is to apply, for instance, it must be shown to be both functional (Eq. 2) and total (Eq. 3):

$$\forall\, a_1, a_2 : A,\ c : C \mid \alpha(c, a_1) \wedge \alpha(c, a_2) \Rightarrow \mathit{equal}(a_1, a_2) \tag{2}$$

$$\forall\, c : C \mid \exists\, a : A \mid \alpha(c, a) \tag{3}$$

What would it mean to check these in Alloy? The three equations above are all expressions of first order logic, and yet they present different levels of difficulty

to the Alloy Analyzer, the model-finding tool supporting the formalism. Equation 3 in particular, which checks whether or not a relation is total, is problematic because its SAT encoding results in an unbounded universal quantifier [9].

Similar checks are required if we have concerns, as we should, about *progress* properties: a concrete operation op_c can "do nothing" and satisfy the correctness check vacuously, e.g., when the term $op_c(c, c')$ is false in Eq. 1, as it might be due to an inadvertently buggy specification. So we add to the set of proof obligations a progress check:

5. *Progress of the concrete operation* – with respect to the operation, every initial state satisfying the concrete precondition must have a corresponding final state.

As with Eq. 3, which requires that α be total, this kind of check introduces an unbounded quantifier in its formulation, and is again problematic for Alloy.

All this points to a limitation of finite instance finding. Various approaches have been devised to try and circumvent it, including the definition of generator axioms [9], though they are sometimes difficult to come by or too computationally expensive to employ, as we later show. Below we describe a new, simpler approach for performing these and other checks in Alloy and do so in the context of HPC.

5 A Parallel HPC Application in Alloy

We illustrate our approach with Alloy models of a parallel program originally specified by Martin [11]. We extend his work by formalizing coarrays in Alloy, adding necessary and sufficient conditions for checking refinement, and defining the state-space invariants required to formally verify them. Our models are available online [1].

Iterative Jacobi Computation. Martin considers an example of the numerical solution to Laplace's equation over a rectangular domain, with fixed values on the boundary, using the technique of Jacobi iteration. The example is implemented in Coarray Fortran (CAF), a single program, multiple data (SPMD) extension to the language. In CAF, designated variables are extended with a parallel dimension, so that each is shared across copies of the same program (images) using the Partitioned Global Address Space (PGAS) model.

At each iteration, the algorithm averages the four-nearest neighbors of all interior elements of a matrix. Because it updates or "displaces" all of the elements at the same time, the Jacobi method is sometimes called the *method of simultaneous displacements,* which contrasts with the Gauss-Seidel method, whose elements are successively "updated in place." As a result, extra storage is needed in the Jacobi method to take a step, but, afterward, the previous step's storage can be reused if we swap matrix storage locations at each iteration.

To specify the parallel program, Martin takes a refinement perspective, defining a step in a sequential Jacobi iteration as the abstract level, and a step in

a parallel CAF implementation as the concrete level. The abstraction relation maps coarrays at the concrete level (a sequence of image matrices) to a single abstract matrix. Duplicating columns at the image interfaces allows computation and then communication to proceed in separate "stages" in the CAF program.

Figure 2 shows the column mapping between coarray images and the abstract matrix (left) and the role of invariants (right) which we use to tighten the abstraction relation α so that it is total. In the concrete space, the invariant enforces interface conditions between neighboring coarrays, i.e., the duplication of columns necessary for halo or border exchanges, and it ensures that the matrices corresponding to a given coarray variable all have the same dimensions in each image, as dictated by CAF semantics. In the abstract space, another invariant enforces basic matrix index and bounds checking.

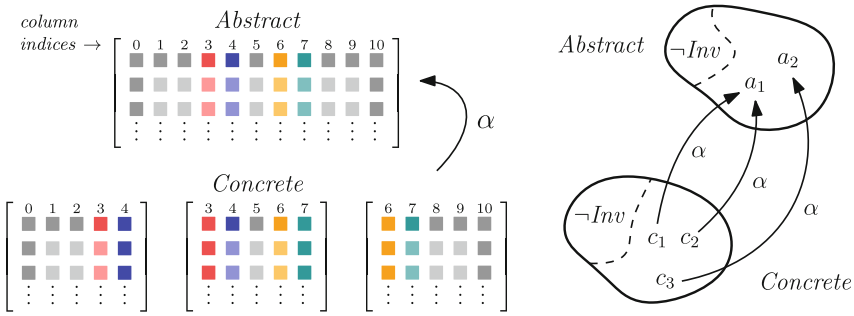

Fig. 2. Mapping coarray images to a matrix (left) and enforcing invariants (right)

Alloy Models and Extensions. We formalize the problem in Alloy using the same matrix structure as Dyer et al. [5], along with new machinery to capture the relevant aspects of Coarray Fortran.

```
sig Matrix {
  rows, cols: Int,
  vals: Int → Int → lone Value
}
```

```
sig Value {}
sig Coarray {
  mseq: seq Matrix
}
```

A *signature* in Alloy introduces both a type and a set of uninterpreted atoms, and may introduce *fields* that define relations over them. A summary of the language is available online (see `alloytools.org/download/alloy-language-reference.pdf`).

Here, intuitively, matrices are defined with two-dimensional indexing, and coarrays are defined as a sequence of matrices. Since Alloy provides no means of representing reals or floating point values, matrix elements are modeled as a number of distinct values, depending on scope size. This simple approach suffices for representing the structural properties of matrices, and where more is needed, arithmetic expressions can be built up and checked symbolically [5].

After adding abstract and concrete Jacobi operations, refinement checks can be performed. Below we describe some of the issues that arise when attempting to verify such obligations, along with their resolution, including a new idiom for checking whether or not a relation is total. Further details and other refinement checks appear in models that are available online [1].

Relative Scope Sizes. Alloy has a rich notion of scope that allows users to bound each signature separately. This accommodates, within a model and across the model's signatures, individual scope sizes that are problem dependent and tailored to the domain. With matrices, however, their sizes are naturally determined by row and column dimensions, a numerical quantity that is bound by a single bitwidth specification in Alloy, which sets the scope of all integers [12]. While otherwise not a concern, when a model calls for matrices of relatively different sizes—like coarray images that correspond to a larger abstract matrix—some checks may produce spurious counterexamples.

In models that relate coarray images and matrices by an abstraction function, like the check in Eq. 2 that determines whether α is functional, we limit the dimensions of coarray images in quantifiers so that the dimensions of their corresponding abstract matrix, if they have one, will remain in scope. Numerically, the relationship is as follows: in Fig. 2 we show a mapping from coarrays with n_c columns and i images, say, to an abstract matrix with n_a columns. When $n_a \geq 4$, column sizes are related as follows: $n_a = i(n_c - 2) + 2$, because of the duplication of border columns. We capture the limit on coarray dimensions in Alloy by defining a bounded, scope-complete subset of coarrays called `CoarraySmall` that can be used as a drop-in replacement when expressions are quantified over coarrays and abstract matrices.

Total Relation Check. For checking whether or not α is total, we must contend with the unbounded universal quantifier problem [9]. To do so, we present a novel approach that avoids the need for generator axioms, which are in any case impractical in this context, because they produce a combinatorial number of instances, namely $\mathcal{O}(v^{n \times n})$ for $n \times n$ matrices whose elements each have v possible values. We do so by adding problem structure, i.e., by introducing an additional level of indirection and reformulating the check:

```
sig P {
  con: Coarray,
  abs: lone Matrix
}
check isTotal { all p: P | alpha[p.con, p.abs] ⇒ some p.abs }
```

Here, instances of `P` necessarily hold a single coarray in `con` and *may* hold a matrix in `abs` that is related to it by the abstraction function `alpha`; the `lone` keyword in Alloy (less than or equal to one) allows the `abs` field to be empty. Intuitively, Alloy searches within a given scope for a counterexample in which a concrete object exists but there is no abstract counterpart, according to `alpha`. If no counterexample can be found, the check is valid within the specified scope.

To show that the approach is equivalent to adding generator axioms, we reformulate the check above as a predicate $\texttt{isTotal}_p$, define another predicate $\texttt{isTotalNaive}_p$ that is equivalent to Eq. 3, and then compare them after including a predicate for the generator axiom:

$$\texttt{check isTotal}_p \Leftrightarrow (\texttt{isTotalNaive}_p \texttt{ and generator})$$

The check passes, though with sizes limited to just 2×2 matrices and four values, since `generator` explodes the scope. In cases where generator axioms cannot be circumvented, as `isTotal` manages, parametric reasoning may be considered [6].

Interleaving Specifications. The communication step, as given in CAF by Martin, makes a subtle but important design choice. A coarray image shares the values it computes in interface columns before the next iteration begins, yet no synchronization barrier is needed. Instead of "pulling" values—which may or may not have been computed—from adjacent matrices, an image "pushes" its computed values by writing to its neighbors. Doing so guarantees the absence of race conditions, and eliminates the need for interleaving-style specifications.

Although we can and have used interleaving to detect race conditions in simple CAF models, it is interesting to ask what happens in applications like that of Martin, which are not written in an update-in-place style, and where there is nevertheless interference, such as the inadvertent overwriting of values by processes due to a bug. Can we find it? In such a case, overwriting produces a contradiction in the antecedent of the correctness check, so it appears safe. Therefore, one needs both safety and progress checks, which we include. That is to say, interference of this kind manifests as lack of progress, which is detectable.

6 Conclusions and Future Work

We present an approach for checking data refinement in Alloy to verify correctness properties of HPC programs. Unlike attempts at after-the-fact verification, our emphasis is on "design thinking," an inherently iterative process that, with tool support, may help practitioners gain a deeper understanding of the structure and behavior of the programs they create. Tangible artifacts from the process include representation invariants that must be maintained by concrete implementations—in languages like Fortran, C/C++, and Julia—and abstraction relations that define and document how they should be interpreted.

Although we believe this to be a promising approach for HPC verification, the work presented in this paper also exposes some of the limitations of finite instance finding in dealing with existential quantifiers, integer bounds, and scope explosion. Practitioners will likely encounter these issues themselves, so developing a common approach for tackling them is necessary. Further work is needed to understand the best Alloy formulations for typical refinement checks as well as addressing technical limitations of the Alloy Analyzer in scientific applications.

Acknowledgments. This work was funded by NSF under the Formal Methods in the Field (FMitF) program, awards #2124205 (NCSU) and #2124100 (Utah).

References

1. Alloy models from the paper. https://go.ncsu.edu/alloy/
2. Baugh, J., Altuntas, A.: Formal methods and finite element analysis of hurricane storm surge: a case study in software verification. Sci. Comput. Program. **158**, 100–121 (2018)
3. Beckingsale, D.A., et al.: Raja: portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81 (2019)
4. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014)
5. Dyer, T., Altuntas, A., Baugh, J.: Bounded verification of sparse matrix computations. In: Proceedings of the Third International Workshop on Software Correctness for HPC Applications, Correctness 2019, pp. 36–43. IEEE/ACM (2019)
6. Emerson, E.A., Trefler, R.J., Wahl, T.: Reducing model checking of the few to the one. In: Liu, Z., He, J. (eds.) Formal Methods Softw. Eng., pp. 94–113. Springer, Berlin, Heidelberg (2006)
7. Gopalakrishnan, G., et al.: Report of the HPC Correctness Summit, 25–26 Jan 2017, Washington, DC. CoRR abs/1705.07478 (2017)
8. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**(4), 271–281 (1972)
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)
10. Lustig, D., Wright, A., Papakonstantinou, A., Giroux, O.: Automated synthesis of comprehensive memory model litmus test suites. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 661–675. ASPLOS 2017, ACM, New York, NY, USA (2017)
11. Martin, J.M.R.: Testing and verifying parallel programs using data refinement. In: Communicating Process Architectures 2017 & 2018, pp. 491–500. IOS Press (2019)
12. Milicevic, A., Jackson, D.: Preventing arithmetic overflows in Alloy. Sci. Comput. Program. **94**, 203–216 (2014)
13. de Roever, W.P., Engelhardt, K., Buth, K.H.: Data refinement: model-oriented proof methods and their comparison. Cambridge University Press (1998)