Verification of the Quotient Graph construction from the molecule

Yaroslav Rozdobudko

June 27, 2025

Contents

1 Introduction

To make sure that our software works correctly we need to verify it. Usually it is done by testing, but it is not the best solution and in some cases, like critical software it is not sufficient. Quotes by Edsger W. Dijkstra show why[1]:

> *"Testing shows the presence, not the absence, of bugs."* — Edsger W. Dijkstra [6]

> *"Program testing can be used to show the presence of bugs, but never to show their absence!"* — Edsger W. Dijkstra [11]

It is impossible to came up with all possible test cases, so there might be some inputs that will break the program. But this limitation can be solved by formalizing programs, and introducing proof that program does what it is intended. Formal verification rigorously proves correctness of program.

The earliest usage of formal methods was by Martin Davis[8, 9, 21] in his proof of the program for Presburger's algorithm[22]. In 1969 Hoare developed *Hoare triples*[16] using Floyd logic[12]. *Hoare triple* is of a form $\{P\}C\{Q\}$ where P is the precondition that is required for the program, C is the command after execution of which third part of the triple will be satisfied[2]. The same paper also provided rules and axioms for work with *Hoare logic*.

Over the years multiple solutions were developed that tried to formalize algorithm. The Boyer-Moore Theorem Prover *Nqthm*[5] and *ACL2* proof assistants, automated proof assistant *Isabelle*, languages *Agda*, *Ada/SPARK*, *Dafny*[18], *Lean*[3]. Formal methods have been used in critical software, hardware design, and also by mathematicians, to help with formal proofs[4].

2 Dafny

2.1 Introduction to Dafny

Dafny is an imperative language with functional and object oriented features[18], and also automated verifier[15] based on *dynamic frames*[17]. Dafny verifier will transform source code to the *Boogie*[1] language, than from *Boogie* first-order verification conditions are generated and passed to *Z3* SMT solver[10] as a query. There is also an option to specify custom SMT solver `--solver-path` but it is not well documented and there is no support beside *Z3* solver.

Dafny binaries can be downloaded from the GitHub repository or build from the source code[5], also can be installed as part of the VS code extension. Dafny `verify` command takes input Dafny source code with extension `dfy`, and will

[1]Quotes by Dijkstra: https://en.wikiquote.org/wiki/Edsger_W._Dijkstra
[2]Explained by Hoare: https://www.youtube.com/watch?v=czzp8gMESSY
[3]History of program verification: https://www.youtube.com/watch?v=HJkukhoQFzo
[4]The Future of Mathematics https://youtu.be/Dp-mQ3HxgDE?si=hFXAZQEgIvmNDjFb
[5]Dafny source code: https://github.com/dafny-lang/dafny

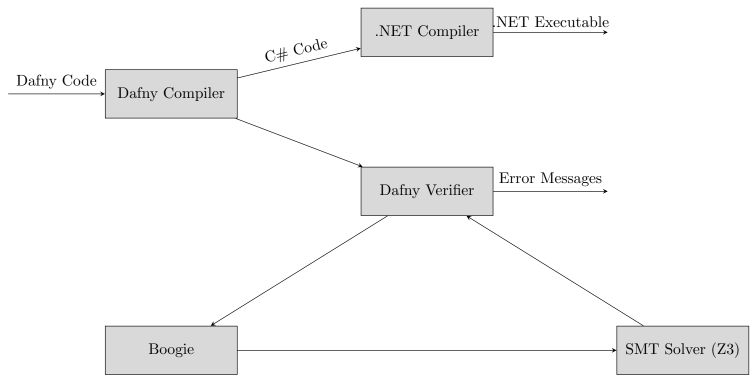


Figure 1: The Dafny system, figure taken from [15].

output results of the verification. It also supports multiple options like increasing time out for verification or providing more logs about proofs. Separate command can be used to compile Dafny to a number of languages (C#, Java, Go, JavaScript, C++, Python). There is an also introduction of the Intermediate Representation (IR) which is intermediate language which will allow to compile Dafny to many other languages, and plans to create compiler that will allow to run Dafny code natively[6]. It is also possible to translate to *Boogie* code using `-print` command and run it with the *Boogie*. Even the simplest Dafny program in *Boogie* will contain several thousand lines, because Dafny includes all the specifications needed for the language itself.

Listing 1: Dafny Triple method

```
method Triple(x: int) returns (r: int) {
    var y := 2 * x;
    r := x + y;
    assert r == 3 * x;
}
```

Listing 2: Boogie Triple method, part of the code translated from Dafny to Boogie

```
...
implementation {:smt_option "smt.arith.solver", "2"}
               {:verboseName "Triple (correctness)"}
               Impl$$_module.__default.Triple(x#0: int)
               returns (r#0: int, $_reverifyPost: bool)
{
  var $_ModifiesFrame: [ref,Field]bool;
  var y#0: int;

```

[6]I heard this news in Dafny video: https://youtu.be/OdTiTiuUMto?si=EuyKXzHr36UTAHCF

```
    // AddMethodImpl: Triple, Impl$$_module.__default.Triple
    $_ModifiesFrame := (lambda $o: ref, $f: Field ::
      $o ≠ null && $Unbox(read($Heap, $o, alloc)):
      bool ⇒ false);
    $_reverifyPost := false;
    // ----- assignment statement ----- Triple.dfy(2,11)
    assume true;
    assume true;
    y#0 := Mul(LitInt(2), x#0);
    // ----- assignment statement ----- Triple.dfy(3,7)
    assume true;
    assume true;
    r#0 := x#0 + y#0;
    // ----- assert statement ----- Triple.dfy(4,4)
    assume true;
    assert {:id "id2"} r#0 == Mul(LitInt(3), x#0);
}
...
```

2.2 How proof works

Dafny program stored in a text file with extension `dfy`. We can separate program in the modules with `module` keyword, and import it in another module. There are multiple built in types: `bool`, `int`, `nat`, etc. In addition to regular operators boolean type supports equivalence (if and only if) `<==>` and implication `==>` operators. Dafny numerical types do not have upper bound, since it is supposed to represent mathematical numbers (practically it will of course will be limited by the computer hardware). There are collection types: sets, sequences, maps etc. Dafny supports *quantifiers*, universal `forall forall x :: P(x)` which is the same as $\forall x : P(x)$, for all x P holds, and existential quantifier `exists exists x :: P(x)` which is the same as $\exists x : P(x)$, there exists x for whom P holds.

Maps and sets support comprehensions, which can be used to create new maps and sets. Similar to mathematical notation $\{f(x)|R(x)\}$, comprehension `set x` R :: f(x)— will define the set of all elements $f(x)$ such that R holds. For example we can define a set of squares of integers from 0 to 9 with:

Listing 3: Set comprehension

```
set x | 0 ≤ x < 10 :: x * x
```

In Dafny **methods** can take variable number of in parameters and return variable number of out parameters. Methods can be used in statements. Functions can be used in expressions, can take variable number of in parameters but return only one type. We can create *proof obligations* using **assert** statement, or by specifying *preconditions* using **requires** keyword and *postconditions* using **ensures** keyword. Dafny tries to prove *proof obligations* automatically. We can define **datatypes** to represent data structures, they do not change state, and can be defined recursively.

To verify that program is correct we need to confirm all the specifications that we used to annotate the program are correct, essentially we make sure that programs adheres to the contract that we specified. By default Dafny will try to verify *proof obligations* automatically, but if it can't it we need to provide additional steps to confirm that *proof obligation* holds. There are multiple ways to prove in Dafny[7]: *lemma*, *assert/assert by*, *calc*. *Assert* statement verifies that a logical preposition is true. Results of the *assert* can be used to prove other *proof obligations*. It is possible to add additional proof steps to *assert* with *by*. *Calc* statements uses *program-oriented calculations* (poC) [20] and allows us to use list of related expressions, to provide intermediate steps for a proof.

Listing 4: Calc statement, taken from[19]

```
calc {
5 * (x + 3);
== // distribute multiplication over addition
5 * x + 5 * 3;
== // use the arithmetic fact that 5 * 3 == 15
5 * x + 15;
}
```

We can reuse program proofs by combining them into *lemmas*, and treat them as an auxiliary theorems that we can use in the proofs, it supports preconditions and postconditions and is a *ghost* method. All the mentioned *proof* tools like *assert* and *calc* are *ghost* constructs, this means that they are not present when we compile Dafny program into an executable program. We can define *ghost* variables or methods ourselves using `ghost` keyword.

Dafny supports *for* and *while* loops. *While* loop specification consist of *quard* and *invariant*. *Guard* specifies until what state loop should continue, *invariant* restricts states where loop can operate. *Guard* checked before each iteration, *invariant* is checked before loop starts, and after each execution of the loop. By the end of the execution loop will reach state where the *Guard* is false and *invariant* still holds. To prove termination, loops support *decreases* statement.

Listing 5: While loop

```
method m(){
  var i := 10;
  while 0 < i
    invariant 0 ≤ i
    decreases i
  {
    i := i - 1;
  }
}
```

[7]Different ways to prove: https://leino.science/papers/krml276.html

Here we will check if *invariant* holds, and our *guard* is true, and loop will be executed until i reaches 0, at this state, *guard* is false but invariant still holds. If we change our invariant to:

Listing 6: Incorrect invariant

```
invariant 2 ≤ i
```

We will get an error:

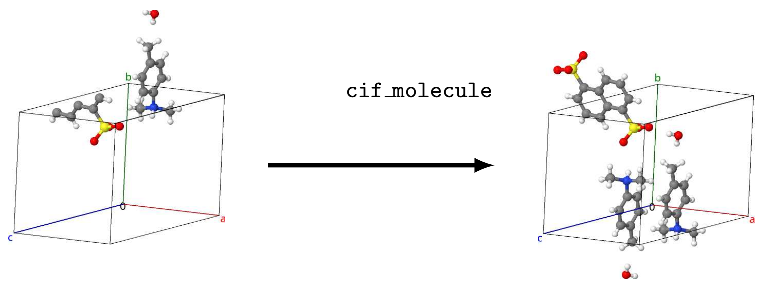
Listing 7: Error on loop invariant

```
 Related message: loop invariant violation
  |
4 |     invariant 2 ≤ i
  |
```

This is because we cannot guaranty that loop *invariant* holds, since i will be 0 by the end of the loop execution.

3 Problem

It is important to reconstruct molecule from the *asymmetric unit* (AU) preserving stoichiometric ratios. To handle this task program *cif_molecule*[14] were developed as part of *cod-tools* package.

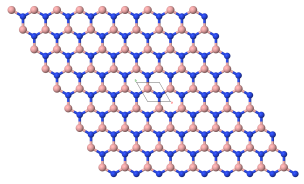


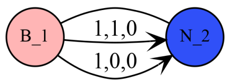
Asymmetric unit *Stoichiometrically correct molecule*

There are special kind of molecules, *polymeric* molecules. Technical term is *coordination polymers*, and there is a long history of problems with definitions [4, 2, 3]. For our purposes we will consider *polymer* an infinite molecule, which can be represented by an infinite repeated net. Infinite nature of *polymers* presents problem on how to create a finite representation of such molecule. There is an algorithm[7] to solve this problem. It uses *quotient graph* to describe all translational equivalent bonds and atoms. *Quotient graph* algorithm uses notion of *equivalence* to group *equivalent* objects, in our case atoms are *equivalent* if they have the same *label* and *symmetry operation* and algorithm will collect all *equivalent* atoms into groups and add connections (edges) between them, we assign value to the edges.

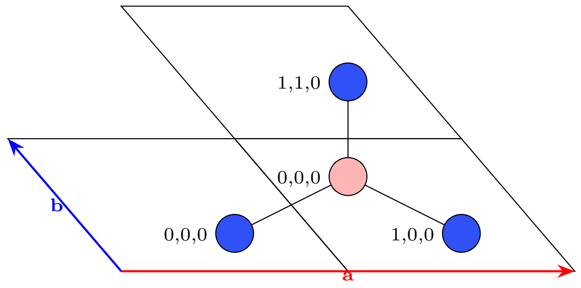
Algorithm to create *Labeled Quotient Graph* (LQG):

1. Choose a coordinate system, select basis vectors.
2. Assign initial labels to the original vertices $(0, 0, 0)$.
3. If the net has an edge from point $P(m, n)$ to point $R(p, q) \Rightarrow$ there is a directed edge in LQG with label $(p - m, q - n)$ from P to R.


Infinite net (COD ID 9008997)



Labeled Quotient Graph (COD ID 9008997)



Constructing Labeled Quotient Graph (LQG)

On the left you can see *honycomb* net, at the center is a visual representation of the construction of LQG and on the right is the result — LQG, vertex *B_1* represents all the *Boron* atoms with symmetry operation 1, and vertex *N_2* represents all the *Nitrogen* atoms with symmetry operation 2, 3 edges between these vertices represent all the possible translations that atoms can take, and can be used to reconstruct molecule.

LQG algorithm was implemented in *cif_molecule* and released in *cod-tools* release *3.11.0*[23]. *cif_molecule* were used on all the entries in the *Crystallography Open Database* (COD)[13].

In addition to multiple tests that were already present, number of tests were added to the *cif_molecule*. But it is impossible to predict all the possible test cases. This potentially makes this algorithm a good target for creating formal specification that can be proved and improve confidence in the algorithm.

4 Implementation

The first step is to define all the datatypes and functions in a common module so we can reuse it. Translation is just a vector that represents atom translation. Atom consist of atom label, *sym* — symmetry operation, and translation. Vertex contains just atom label and symmetry operation, which what we use to define *equivalence* for atoms. Function *Id* returns vertex from an atom, and *AllNodeIds* returns all the Ids (vertices) from sequence of atoms using set comprehension.

Listing 8: Common module

```
module Common {
  datatype Translation = Translation(x: int, y: int, z: int)
  datatype Atom    = Atom(atomLabel: string, sym: int,
                         translation: Translation)

  datatype Vertex = Vertex(atomLabel: string, sym: int)
  datatype Bond = Bond(a: Atom, b: Atom)
  datatype Edge = Edge(u: Vertex, v: Vertex,
                       shift: Translation)

  function Id(a: Atom): Vertex { Vertex(a.atomLabel, a.sym) }

  function AllNodeIds(atoms: seq<Atom>): set<Vertex> {
    set i | 0 ≤ i < |atoms| :: Id(atoms[i])
  }

  function Subtract (u: Translation, v: Translation):
                Translation {
                    Translation(u.x-v.x, u.y-v.y, u.z-v.z)
                }
}
```

Now we can implement the algorithm for constructing *LQG*. We need to simplify input and output, method will accept sequences of atoms and bonds, and will return sets of vertices and edges which will represent *LQG*.

Listing 9: ConstructLQG specification

```
method ConstructLQG(atoms: seq<Atom>, bonds: seq<Bond>)
returns (V: set<Vertex>, E: set<Edge>)
  requires |atoms| > 0
  requires forall b :: b in bonds ==> b.a in atoms &&
                                      b.b in atoms
  ensures forall e1,e2 | e1 in E && e2 in E && e1 != e2 ::
  !(e1.u == e2.v && e1.v == e2.u &&
    e1.shift == Negate(e2.shift))
  ensures forall i | 0 <= i < |atoms| :: Id(atoms[i]) in V
  ensures V == AllNodeIds(atoms)
```

Our only preconditions are that there are atoms, and that they are connected. In postconditions we make sure that all atoms are represented in the vertices, and that no distinct edges are inverses of each other, and that vertices contain all *equivalent* classes from atoms.

We assume that molecule connected and since we already have all the bonds we get all the node ids from atoms, and assign visited. Than iterate over all atoms, add invariant to verify that no opposite edges added.

Listing 10: Outer loop

```
var allNodeIds := AllNodeIds(atoms);
var visited : set<Vertex> := allNodeIds;
var edges : set<Edge> := {};

var atomIdx := 0;
while atomIdx < |atoms|
  decreases |atoms| - atomIdx
  invariant forall e1,e2 | e1 in edges && e2 in edges &&
                                e1 != e2 ::
    !(e1.u == e2.v && e1.v == e2.u &&
        e1.shift == Negate(e2.shift))
{
```

Iterate over all the bonds make sure to not add opposite edge.

Listing 11: Inner loop

```
var currentAtom := atoms[atomIdx];
atomIdx := atomIdx + 1;
var currentNodeId := Id(currentAtom);

var bondIdx := 0;
while bondIdx < |bonds|
  decreases |bonds| - bondIdx
  invariant forall e1,e2 | e1 in edges && e2 in edges &&
                                e1 != e2 ::
  !(e1.u == e2.v && e1.v == e2.u &&
        e1.shift == Negate(e2.shift))
{
```

Only process bonds that have current atom, if bond contains current atom calculate edge label, and if this edge has not been added yet, add it to the set.

Listing 12: Inner loop, body

```
var bond := bonds[bondIdx];
bondIdx := bondIdx + 1;

if bond.a == currentAtom {
  var neighborNodeId := Id(bond.b);
  var shift := Subtract(bond.b.translation,
                        bond.a.translation);
  var edge := Edge(currentNodeId, neighborNodeId, shift);
  var invEdge := Edge(neighborNodeId,
                      currentNodeId,
                      Negate(shift));

  if edge !in edges && invEdge !in edges {
    edges := edges + {edge};
  }
} else if bond.b == currentAtom {
```

Text_Construct module were created in order to test construction of the *LQG*. There we construct molecule that we seen in 3, we add minimal number of atoms and bonds needed to construct *LQG*. After constructing *LQG* auxiliary lemma called to prove number of vertices and that they are in the set.

Listing 13: Test construction of *LQG* from molecule

```
var B0 := Atom("B",1, Translation(0,0,0));
var N00 := Atom("N",2, Translation(0,0,0));
var N10 := Atom("N",2, Translation(1,0,0));
var N11 := Atom("N",2, Translation(1,1,0));

var atoms := [B0, N00, N10, N11];
var bonds := [Bond(B0,N00), Bond(B0,N10), Bond(B0,N11)];

var V, E := ConstructLQG(atoms, bonds);
...

TestCaseNodeIdProperties(atoms);
assert |V| == 2;
assert Vertex("B", 1) in V;
assert Vertex("N", 2) in V;
```

Since it is in the Main method, when compiled this code will print results.

```
Results:
|V| = 2 unique vertices
|E| = 3 edges
Test 0 vertex Common.Vertex.Vertex("B", 1)
Test 1 vertex Common.Vertex.Vertex("N", 2)
Vertex count: 2
Test 0 edge Common.Edge.Edge(Common.Vertex.Vertex("B", 1),
                             Common.Vertex.Vertex("N", 2),
                             Common.Translation.Translation(0, 0, 0))
Test 1 edge Common.Edge.Edge(Common.Vertex.Vertex("B", 1),
                             Common.Vertex.Vertex("N", 2),
                             Common.Translation.Translation(1, 0, 0))
Test 2 edge Common.Edge.Edge(Common.Vertex.Vertex("B", 1),
                             Common.Vertex.Vertex("N", 2),
                             Common.Translation.Translation(1, 1, 0))
Edges count: 3
```

Dafny version *4.10.1* was used for executions. Program were split into modules, and executed using *make*. Default target will run *verify* command with option `--log-format text` and results of the execution will be stored in the `results/verification/verify.log`. Target *build* will run *build* command, build log will be stored in the `results/build/build.log`, directory `results/build/` will store executables.

5 Conclusions

Dafny is a powerful language and a automated program verifier with rich set of features. Unfortunately only simplified version of the algorithm was implemented, and proof is rudimentary. It is hard to come up with a proper postconditions and the proof. Additional method to reconstruct molecule from LQG was implemented but it is very simplified, and does not follows the original algorithm. After more practice and exposure to formal methods I think I will be able to improve my implementation.

References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*, page 364–387. Springer Berlin Heidelberg, 2006.

[2] Stuart R. Batten, Neil R. Champness, Xiao-Ming Chen, Javier Garcia-Martinez, Susumu Kitagawa, Lars Öhrström, Michael O'Keeffe, Myunghyun Paik Suh, and Jan Reedijk. Coordination polymers, metal–organic frameworks and the need for terminology guidelines. *CrystEngComm*, 14(9):3001, 2012.

[3] Stuart R. Batten, Neil R. Champness, Xiao-Ming Chen, Javier Garcia-Martinez, Susumu Kitagawa, Lars Öhrström, Michael O'Keeffe, Myunghyun Paik Suh, and Jan Reedijk. Terminology of metal–organic frameworks and coordination polymers (IUPAC recommendations 2013). *Pure and Applied Chemistry*, 85(8):1715–1724, jul 2013.

[4] Stuart R. Batten and Richard Robson. Interpenetrating nets: Ordered, periodic entanglement. *Angewandte Chemie International Edition*, 37(11):1460–1494, June 1998.

[5] Robert S. Boyer and Yuan Yu. *Automated correctness proofs of machine code programs for a commercial microprocessor*, page 416–430. Springer Berlin Heidelberg, 1992.

[6] J.N. Buxton, B. Randell, and NATO Science Committee. *Software Engineering Techniques: Report on a Conference Sponsored by t he NATO Science Committee, Rome, Italy, 27th to 31st October 1969.* NATO Science Committee, 1970.

[7] S. J. Chung, Th. Hahn, and W. E. Klee. Nomenclature and generation of three-periodic nets: the vector method. *Acta Crystallographica Section A Foundations of Crystallography*, 40(1):42–50, jan 1984.

[8] M. Davis. A computer program for Presburger's algorithm. In *Proving Theorems (as Done by Man, Logician, or Machine)*, 1957. Proc. Summer Institute for Symbolic Logic. Second edition; publication date is 1960.

[9] Martin Davis. The early history of automated deduction. *Handbook of Automated Reasoning*, 1, 03 2002.

[10] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, page 337–340. Springer Berlin Heidelberg, 2008.

[11] E.W. Dijkstra. *Ewd 249 Notes on Structured Programming: 2nd Ed.* Technische Hogeschool Eindhoven. Department of Mathematics, 1970.

[12] I Floyd. Rw, assigning meanings to programs. *MathematiCal Aspects of Computer Science, ed. Schwartz, JT, Amer. Math. Soc*, 1967.

[13] Saulius Gražulis, Adriana Daškevič, Andrius Merkys, Daniel Chateigner, Luca Lutterotti, Miguel Quirós, Nadezhda R. Serebryanaya, Peter Moeck, Robert T. Downs, and Armel Le Bail. Crystallography open database (cod): an open-access collection of crystal structures and platform for world-wide collaboration. *Nucleic Acids Research*, 40(D1):D420–D427, November 2011.

[14] Saulius Gražulis, Andrius Merkys, Antanas Vaitkus, and Mykolas Okulič-Kazarinas. Computing stoichiometric molecular composition from crystal structures. *Journal of Applied Crystallography*, 48(1):85–91, 2015.

[15] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. *Using Dafny, an Automatic Program Verifier*, pages 156–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[17] Ioannis T. Kassios. *Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions*, page 268–283. Springer Berlin Heidelberg, 2006.

[18] K. Rustan M. Leino. *Dafny: An Automatic Program Verifier for Functional Correctness*, page 348–370. Springer Berlin Heidelberg, 2010.

[19] K Rustan M Leino and Kaleb Leino. *Program proofs.* MIT Press, London, England, March 2023.

[20] K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, pages 170–190, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[21] Liesbeth De Mol, Yuri V. Matiyasevich, Eugenio G. Omodeo, Alberto Policriti, Wilfried Sieg, and Elaine J. Weyuker. Martin davis: An overview of his work in logic, computer science, and philosophy, 2025.

[22] Mojzesz Presburger. Uber die vollstandigkeiteines gewissen systems der arithmetik ganzer zahlen, in welchen die addition als einzige operation hervortritt. In *Comptes-rendus du ler congres des mathematiciens des pays slavs*, 1929.

[23] Yaroslav Rozdobudko, Antanas Vaitkus, Andrius Merkys, and Saulius Gražulis. cod-tools, version 3.11.0, 2025. `svn://www.crystallography.net/cod-tools/tags/v3.11.0` Last accessed: 2025-05-08.