NASA/TM-2020-220588

Formal Verification of a Solution to the n-Queens Problem

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

April 2020

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM-2020-220588

Formal Verification of a Solution to the n-Queens Problem

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

April 2020

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

This report describes a formal verification of a concise algorithm that computes a solution to the *n*-Queens problem for all natural numbers n, such that $n > 3$. The formal proof of the algorithm is completed in the Prototype Verification System (PVS) theorem prover. This report illustrates that theorem provers are more capable than model checkers when verifying an algorithm with potentially infinitely many input values using a concise algorithm for a general problem that produces a solution. The particular algorithm used was chosen as a pedagogical example for formal proof that does not employ recursion and is computationally less intensive than the backtracking method.

Keywords: Model Checking, Theorem Proving, PVS, Verification, n-Queens, Formal Proof

Table of Contents

ABSTRACT .... **I**

TABLE OF CONTENTS .... **II**

1. INTRODUCTION **1**

2. THE ALGORITHM **3**

3. VERIFICATION **4**

4. SUMMARY AND CONCLUSION **7**

ACKNOWLEDGEMENT 7

REFERENCES **7**

APPENDIX – PVS SPECIFICATION **9**

1. Introduction

In this report, we introduce formal verification methods and highlight merits of various verification techniques, namely model checking and theorem proving, over testing. We also present a candidate algorithm, its formal specification, and analysis of its proof using a theorem prover.

The eight queens problem is a classic problem using a chessboard. The goal is to place eight queens on an 8×8 chessboard so that no one queen can take another. For those familiar with the game of chess, the eight queens problem is as follows: arrange eight queens on a chess board so that none of them is in check of any other. For those unfamiliar with the game of chess, the problem may be stated as follows: find the different ways to place eight pieces (queens) on the chessboard so that no two of them share the same row, column, or diagonal. The eight queens problem is an example of the more general ***n*-Queens problem** of placing n queens on an $n \times n$ chessboard. Solutions exist for the n-Queens problem for all natural numbers $n > 3$, and there are no solutions for $n \leq 3$ [Spr 1899], [Bal 1960], [Hof 1969].

Although mathematician C.F. Gauss is credited with this problem, the eight queens problem was originally published by German chess composer Max Bezzel in 1848. Many mathematicians have since worked on this problem. The first solution was published by Franz Nauck in 1850. He also extended the eight queens problem to n-Queens [Bal 1960].

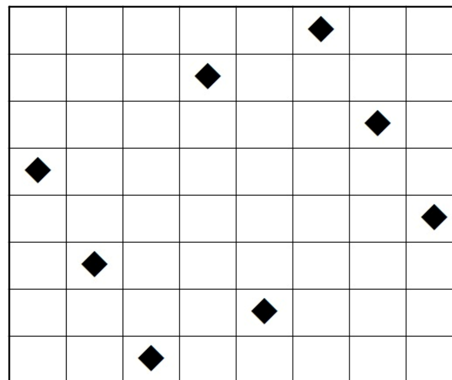

Figure 1. A solution to the eight queens problem.

For the eight queens there are 92 distinct solutions, one of which is shown in Figure 1. However, if solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 unique (or fundamental) solutions.

In [Dah 1972] Dahl et al. mention that Edsger Dijkstra used this problem to illustrate the power of what he called structured programming and published a highly detailed description of a depth-first backtracking algorithm. Backtracking is a well-known exhaustive search method that systematically examines all possible solutions for a queen at a particular position. If a solution is not possible, then the algorithm backtracks to consider other possible positions. The technique dates back at least 100 years and has since been successfully applied to a variety of problems, e.g., Artificial Intelligence (AI) and Sudoku. Some of these algorithms are used to produce a

solution for the *n*-Queens problem for a given *n* while others are aimed at producing all solutions for a given *n*.

There are a number of solutions to the *n*-Queens problem. It appears that the most commonly cited solutions that are formally verified are essentially variations of the backtracking algorithm [Fil 2012], [Kea 2000].

There are also explicit algorithms for placing *n* queens that require no combinatorial search whatsoever [Ber 1991], [Hof 1969]. The particular algorithm we chose to verify uses algebraic operations [Ber 1991]. It does not employ recursion and does not require an exhaustive search. Thus, this algorithm is computationally less intensive than the backtracking method. We describe the algorithm in the following section.

The purpose of this report is not to present a new algorithm for finding a solution for a given *n*. Nor is the purpose to count all possible solutions for a given *n*. Rather, the purpose is to formally verify an existing algorithm that guarantees a solution for any given *n*; thus, highlighting the capability of theorem provers in handling arbitrary values. Hoffman et al. [Hof 1969] provided a paper-and-pencil proof of their proposed solution to this problem in their report. However, verification of correctness of an algorithm by the composition of a paper-and-pencil proof and/or manual examination of the proof is error prone [Mal 2006]. Although we believed the given proof was correct, to our knowledge, its correctness had not been formally verified.

There are two general formal methods approaches for the verification of the correctness of an algorithm: **theorem proving** and **model checking**. Theorem proving requires a deductive proof of the protocol. Model checking is used for its ease, feasibility, and quick examination of the problem before a more rigorous attempt at proof is undertaken by a theorem prover. Model checking is based on specific scenarios and is generally limited to a subset of the problem space, which helps to highlight a key difference between theorem proving and model checking. Namely, model checking struggles to verify algorithms that can have a possibly infinite number of inputs. That is, for some systems, testing or model checking alone cannot possibly give a 100 percent guarantee of correctness. This limitation is not present with interactive theorem provers like Prototype Verification System (PVS) [Owr 2008], [Sha 1999], [Owr 1992], as is illustrated by the solution to the *n*-Queens problem. PVS allows a user to input a proof of a mathematical statement, which it checks for logical correctness. Thus, the verification of this algorithm in PVS illustrates that, in more complicated problems when the correctness property is essential, such as a safety-related condition, there is an advantage to using an interactive theorem prover rather than testing or model checking. If the system itself has an infinite number of input values, it is typically impossible to guarantee that, for a given algorithm, a particular property, and for safety critical systems, a safety property, holds by using testing or model checking, but it is possible when using an interactive theorem prover.

The number of fundamental solutions for *n* increases exponentially, for instance, for $n = 8$ and 27, the number of fundamental solutions are 12 and 29,363,495,934,315,694 (or $> 29 \times 10^{15}$), respectively. Thus, as *n* increases, when using a model checker, the required memory, computing power, and time it takes to verify all solutions becomes impractical. Verification via a theorem prover like PVS, however, is independent of the actual value of *n*.

A formal verification of a two-line C program that computes the number of solutions to the *n*-Queens problem using a variety of tools, e.g., Why3, Alt-Ergo, CVC3, and Coq, was reported in [Fil 2012]. The author emphasized that even the shortest program can be a challenge for formal verification. In [Fil 2012] two kinds of integer overflows are reported, depending on the use of integers as bit vectors or as counters. The count of solutions for $n \leq 19$ would overflow with 32-bit integers. Similarly, the count of solutions for $n \leq 28$ would overflow with 64-bit integers. These limitations are absent in PVS, which uses unbounded integers.

This report is organized as follows. Description of the algorithm for finding the solution to the *n*-Queens problem is presented in Section 2. Section 3 is a description of the verification of the algorithm in PVS. Section 4 is a summary of the work and concludes the report. The PVS specification of the solution is presented in the Appendix.

2. The Algorithm

The *n*-Queens algorithm presented here is attributed to Bo Bernhardsson [Ber 1991] who, in turn, based his algorithm on an earlier work by Hoffman et al. [Hof 1969]. The solution reported in [Hof 1969] is given by the three constructions listed below. In 1991, Bernhardsson brought attention to the solution provided by Hoffman et al. to point out that there is a much faster way to find a solution to the *n*-Queens problem [Ber 1991] than a polynomial time algorithm. This explicit solution in fact requires very little computer time and thus makes the *n*-Queens problem a bad benchmark problem for the purpose of comparing computational efficiency. It does however provide a great example highlighting the capability of theorem provers like PVS compared to model checkers in handling algorithms with potentially infinitely many input values. The following solutions can be found in [Ber 1991]. This algorithm is not a backtracking algorithm but is an explicit one.

Let (i, j) be the square in column i and row j on the $n \times n$ chessboard, k an integer and for $i = 1, 2, ..., floor(n/2)$. Then,

1. If n is even and $n \neq 6k + 2$, then
   place queens at $(i, 2i)$ and $(n/2 + i, 2i - 1)$.
2. If n is even and $n \neq 6k$, then
   place queens at $(i, 1 + (2i + n/2 - 3 \ (mod\ n)))$ and $(n + 1 - i, n - (2i + n/2 - 3 \ (mod\ n)))$.
3. If n is odd, then
   use one of the patterns above for $(n - 1)$ and add a queen at (n, n).

Note that $n \neq 6k + 2$ means that $n \ mod\ 6 \neq 2$, and similarly, $n \neq 6k$ means that $n \ mod\ 6 \neq 0$. This algorithm produces a solution for any natural number $n > 3$, and a paper-and-pencil proof of it is provided in [Hof 1969]. We would like to emphasize that this algorithm does not produce all possible solutions or a count of all possible solutions for a given n.

The above algorithm is translated into PVS and named *FinalChess*:

```
FinalChess (n : posnat | n > 3) : ChessType (n) =
  IF even? (n) AND (mod (n, 6) /= 2) THEN ChessMethod1 (n)
  ELSIF even? (n) THEN ChessMethod2 (n)
  ELSIF (mod (n - 1, 6) /= 2) THEN
    (LAMBDA (j : subrange (1, n)) : IF (j = n) THEN n ELSE ChessMethod1 (n - 1)(j) ENDIF)
  ELSE (LAMBDA (j : subrange (1, n)) : IF (j = n) THEN n ELSE ChessMethod2 (n - 1)(j) ENDIF)
  ENDIF
```

In the above description, together, functions *ChessMethod1()* and *ChessMethod2()* address steps 1 through 3 of the solution.

The correctness of the above algorithm, i.e., verification that it produces a solution to the *n*-Queens problems for all $n > 3$, is the main claim and is captured by the following theorem:

```
QueensFinal : THEOREM FORALL (n : {t: posnat | (t > 3)}) :
  QueensSolution (n, FinalChess (n))
```

3. Verification

In this section, we present our formal verification and proof of the *n*-Queens algorithm described in Section 2 in PVS. We present our proof in terms of an object, *chess*, which is an array of size *n* representing an $n \times n$ chessboard. Each element of the array is a positive natural number ranging from 1 to *n*. The *chess* object is filled with the solution produced by the algorithm and each element of the *chess* object contains the column assigned to the queen for the corresponding row. The PVS specification of the algorithm is made available in the appendix.

The algorithm described in Section 2 is elegant and deceptively simple. Although readily described in three lines, it imposed a number of unforeseen challenges in the formal proof/verification process. This phenomenon has been acknowledged by the practitioners of various mechanical verifiers [Fil 2012], [Mal 2012].

```
ChessType (n : posnat) : TYPE = ARRAY [subrange (1, n) → {x : posnat | x ≤ n}]
```

The algorithm is described by three cases. The first two cases deal with even *n*, which, in the description of the proof process, we refer to as methods 1 and 2. Each of these cases in turn is divided into two distinctive regions: $i = 1, 2, ..., floor(n/2)$ and $j = floor(n/2), ..., n$.

To prove that an algorithm produces a correct solution to the *n*-Queens problem, we must prove that no two queens are on the same row, column, or diagonal, and thus cannot attack each other. The following lemmas help to examine whether any two queens are on the same row or column of the chessboard:

```
M1Perm1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i, j : subrange (1, n)) :
  (i ≠ j) IMPLIES (2 * i) ≠ (2 * j - 1)

M2Perm1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 0)}, i, j : subrange (1, n / 2)) :
  (i ≠ j) IMPLIES (1 + (mod (2 * i + n / 2 - 3, n))) ≠ (1 + (mod (2 * j + n / 2 - 3, n)))

M2Perm2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 0)}, i, j : subrange (1, n / 2)) :
  (1 + (mod (2 * i + n / 2 - 3, n))) ≠ (n - (mod (2 * j + n / 2 - 3, n)))
```

The algorithm is easy to follow and simple to implement. It is well structured and treats even and odd numbers separately, with odd numbers as a special case. However, mechanical/formal verification of this algorithm has proven to be very difficult. One of the main challenges in the verification process was dealing with the *mod* (modulo) operator. Since this operation was an essential part of the algorithm and appeared many times in the proof of various lemmas, we chose to unroll it into a concrete value, splitting the proof into multiple cases which had to be handled separately. The following two lemmas unroll *mod* for the two distinct regions that are associated with even values of *n*. It may be useful for others who want to formalize a solution to the *n*-Queens problem to realize that having the following concrete values as the output of the modulo function makes it easier to work with than the modulo function itself.

```
UnrollMod1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i : subrange (1, n / 2)) :
  mod (2 * i + n / 2 - 3, n) =
    IF ((2 * i + n / 2 - 3) < n) THEN (2 * i + n / 2 - 3)
    ELSE ((2 * i + n / 2 - 3) - n)
    ENDIF

UnrollMod2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i : subrange (n / 2 + 1, n)) :
  mod (2 * (i - n/2) + n / 2 - 3, n) =
    IF ((2 * (i - n/2) + n / 2 - 3) < n) THEN (2 * (i - n/2) + n / 2 - 3)
    ELSE ((2 * (i - n/2) + n / 2 - 3) - n)
    ENDIF
```

Another issue was examining whether any two queens were able to attack each other diagonally. We used the slope equation (gradient of a line) for this purpose. A slope of ±1 between their positions, or alternatively, *abs (slope) = 1*, would indicate that the two queens were able to attack each other diagonally:

```
DiagonalCheck (X1, Y1, X2, Y2 : posnat) : bool = ((X2 ≠ X1) AND abs (((Y2 - Y1) / (X2 - X1))) = 1)
```

The following lemma, proven in PVS, uses the fact that the $n \times n$ chessboard is symmetric to allow the inputs of the *DiagonalCheck* function to be readily swapped. As stated earlier, solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one.

```
DiagonalCheckSym : LEMMA FORALL (X1, Y1, X2, Y2 : posnat) :
   DiagonalCheck (X1, Y1, X2, Y2) = DiagonalCheck (X2, Y2, X1, Y1)
```

The following lemmas, which have also been proven in PVS, address various special cases in the proof that the algorithm never produces two queens whose positions satisfy the *DiagonalCheck* function:

```
M1Diag1 : LEMMA FORALL (n : {t: posnat | (t > 3)}, i, j : subrange (1, n)) :
  (i ≠ j) IMPLIES NOT DiagonalCheck (i, 2 * i, j, 2 * j)

M1Diag2 : LEMMA FORALL (n : {t: posnat | (t > 3)}, i, j : subrange (n / 2 + 1, n)) :
  (i ≠ j) IMPLIES NOT DiagonalCheck (i, 2 * (i - n / 2) - 1, j, 2 * (j - n / 2) - 1)

M1Diag3_1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 2)},
  i : subrange (1, n / 2), j : subrange (n / 2 + 1, n)) :
    NOT DiagonalCheck (i, 2 * i, j, 2 * (j - n / 2) - 1)
```

```
M1Diag3_2 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange (1, (n - 1) / 2)) :
  NOT DiagonalCheck (i, 2 * i, ((n - 1) / 2 + i), 2 * i - 1)

M1Diag3_3 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3) AND (mod (t - 1, 6) ≠ 2)},
  i : subrange (1, (n - 1) / 2), j : subrange ((n + 1) / 2, n - 1)) :
    NOT DiagonalCheck (i, 2 * i, j, 2 * (j - (n - 1) / 2) - 1)

M1Diag3_4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i, j : subrange ((n + 1) / 2, n - 1)) :
  (i ≠ j) IMPLIES NOT DiagonalCheck (i, 2 * (i - (n - 1) / 2) - 1, j, 2 * (j - (n - 1) / 2) - 1)

M1Diag4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange (1, (n - 1) / 2)) :
  NOT DiagonalCheck (i, 2 * i, n, n)

M1Diag5 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange ((n + 1) / 2, n - 1)) :
  NOT DiagonalCheck (i, 2 * (i - (n - 1) / 2) - 1, n, n)

M2Diag1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 0)}, i, j : subrange (1, n / 2)) :
  (i ≠ j) IMPLIES NOT DiagonalCheck (i, 1 + (mod (2 * i + n / 2 - 3, n)), j, 1 + (mod (2 * j + n / 2 - 3, n)))

M2Diag2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 0)}, i, j : subrange (1, n / 2)) :
  (i ≠ j) IMPLIES NOT DiagonalCheck (n + 1 - i, n - (mod (2 * i + n / 2 - 3, n)), n + 1 - j, n - (mod (2 * j + n / 2 - 3, n)))

M2Diag3 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) ≠ 0)}, i, j : subrange (1, n / 2)) :
  NOT DiagonalCheck (i, 1 + (mod (2 * i + n / 2 - 3, n)), n + 1 - j, n - (mod (2 * j + n / 2 - 3, n)))

M2Diag4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3) AND (mod (t - 1, 6) ≠ 0)}, i : subrange (1, (n - 1) / 2)) :
  NOT DiagonalCheck (i, 1 + (mod (2 * i + (n - 1) / 2 - 3, (n - 1))), n, n)

M2Diag5 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange ((n + 1) / 2, n - 1)) :
  NOT DiagonalCheck (i, (n - 1) - (mod (2 * (n - i) + (n - 1) / 2 - 3, (n - 1))), n, n)
```

The following predicate examines whether a particular solution, i.e., an element *ct* of *ChessType(n)*, is a valid solution to the *n*-Queens problem:

```
QueensSolution (n : posnat, ct : ChessType (n)) : bool =
  FORALL (i, j : subrange (1, n)) :
    (i ≠ j IMPLIES ct (i) ≠ ct (j)) AND
    (i ≠ j IMPLIES NOT DiagonalCheck (i, ct (i), j, ct (j)))
```

The main result that has been formally proven in PVS is the following theorem, which states that the solution returned by *FinalChess* satisfies the *QueensSolution* condition:

```
QueensFinal : THEOREM FORALL (n : {t: posnat | (t > 3)}) :
  QueensSolution (n, FinalChess (n))
```

Others in the formal methods community [Fil 2012] have used model checking to test that a particular solution to the *n*-Queens problem is valid, but the limitations of model checking exclude a proof in the general case, which states that the particular function returns a valid solution for every possible number *n*. A model checker can only test this for values of *n* less than some threshold. However, the above main theorem, *QueensFinal*, states that the algorithm *FinalChess* returns a correct solution to the *n*-Queens problem for every possible natural number *n* that is at least 4. As an example, the following result has also been proven in PVS by simply testing all possible inputs and in particular without using any of the lemmas stated above. It states the same result as the result above, except in the case where *n* is 8, but the proof is trivial since *n* is a specific value:

```
EightQueens : THEOREM FORALL (n : {t : posnat | (t > 3)}, Chess : ChessType (n)) : QueensSolution (8, FinalChess (8))
```

4. Summary and Conclusion

We introduced the *n*-Queens problem and discussed backtracking, a well-known exhaustive search method that systematically examines all possible solutions for a queen at a particular position. We discussed the merits of model checking and theorem proving in tacking this problem. We presented our reasons for the particular algorithm we chose as a pedagogical example for formal proof as not employing recursion and being computationally less intensive than the backtracking method.

As noted thus far, theorem proving has some advantages over model checking when verifying an algorithm that has an infinite number of possible input values, like the *n*-Queens problem. Model checkers can efficiently test a large input state space, but they are limited when it comes to proving a result for every possible input out of an infinite number of choices. Results such as these typically require a mathematical proof, along with the insight of a human to navigate the proof, which is precisely what a theorem prover like PVS makes possible. Thus, the *n*-Queens problem highlights a difference between model checking and theorem proving, and especially the advantages of theorem proving when the system itself can have an infinite number of input values. As noted in the introduction, the *n*-Queens problem therefore provides a simple example that highlights the sufficiency of a formal proof and how testing and model-checking alone cannot give absolute guarantees of correctness/safety, while this is tractable for a theorem prover like PVS. Another key property that was proven in PVS is the completeness of the given algorithm, namely that unlike backtracking, this algorithm always returns a solution, which is also correct.

Acknowledgement

The author would like to thank Anthony J. Narkawicz, formerly a NASA employee, for his helpful comments and suggestions.

References

[Bal 1960] W. W. Rouse Ball: "*The Eight Queens Problem*", in Mathematical Recreations and Essays, Macmillan, New York, pp. 165–171, 1960.

[Ber 1991] Bernhardsson, B.: *Explicit Solutions to the N-Queens Problem for all N*, SIGART Bulletin, 2, 2, pg 7, April 1991.

[Dah 1972] O.J. Dahl, E. W. Dijkstra, C. A. R. Hoare: "*Structured Programming*", Academic Press, London, 1972 ISBN 0-12-200550-3 see pp. 72–82 for Dijkstra's solution of the 8 Queens problem.

[Fil 2012] Filliatre, J.C.: *Verifying Two Lines of C with Why3: An Exercise in Program Verification*, Lecture Notes in Computer Science Volume 7152, pp 83-97, 2012.

[Kea 2000] Kearse, M.D.; Gibbons, P.B.: *Computational Methods and New Results for Chessboard Problems*, CDMTCS Research Report Series, May 2000.

[Hof 1969] Hoffman, E.J.; Loessi, J.C.; Moore, R.C.: *Constructions for the Solution of the m Queens Problem*, Mathematics Magazine, Vol. 42, No.2, pp 66-72, March 1969.

[Mal 2006] Malekpour, M.R.; Siminiceanu, R.: *Comments on the 'Byzantine Self-Stabilizing Pulse Synchronization' Protocol: Counterexamples*, NASA/TM-2006-213951, February 2006.

[Mal 2012] Malekpour, M.R.: *Model Checking A Self-Stabilizing Synchronization Protocol For Arbitrary Digraphs*, 31st Digital Avionics Systems Conference, October 2012.

[Sha 1999] Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: *PVS System Guide, PVS Language Reference, PVS Prover Guide, PVS Prelude Library, Abstract Datatypes in PVS, and Theory Interpretations in PVS*, Computer Science Laboratory, SRI International, Menlo Park, CA (1999)

[Owr 1992] Owre, S., Rushby, J., and Shankar, N.: *A Prototype Verification System*, in Deepak Kapur, editor, Proc. 11th Int. Conf. on Automated Deduction, volume 607 of Lecture Notes in Artificial Intelligence, pages 748–752. Springer-Verlag (1992)

[Spr 1899] Sprague, T.B.: *On the question of the eight queens problem*, proceedings of the Edinburgh Mathematical Society 17, pp 43-68, 1899.

[Owr 2008] Owre, S.; Shankar, N: *A Brief Overview of PVS,* Theorem Proving in Higher Order Logics (TPHOLS) 2008, in Lecture Notes in Computer Science (LNCS), Volume 5170, pp 22-27, August 2008.

Appendix – PVS Specification

```
% File Name:     Queens.pvs
% Author:        Mahyar R. Malekpour
%                NASA Langley Research Center
%                Hampton, VA 23681-2199
%
%.........................................................................................
Queens : THEORY
BEGIN

  IMPORTING ints@primes, ints@mod_lems

%.........................................................................................
% The data structure.
%
  ChessType (n : posnat) : TYPE = ARRAY [subrange (1, n) -> {x : posnat | x <= n}]

%.........................................................................................
% Generic lemmas.
%
  XPlusOne_grt_X : LEMMA FORALL (X, Y : int) :
    NOT ((((X + 1) / 2) > Y) AND (Y > ((X) / 2)))

  XPlusOne_grt_X_Even : LEMMA FORALL (X, Y : int) :
    even? (X) IMPLIES NOT (((X / 2 + 1) > Y) AND (Y > ((X) / 2)))

  XPlusOne_grt_XMinusOne : LEMMA FORALL (X, Y : int) :
    odd? (X) IMPLIES NOT ((((X + 1) / 2) > Y) AND (Y > ((X - 1) / 2)))

  mod_minus_eq_0_divides : LEMMA FORALL (x, y : nat, n : {t : nat | t > 0}) : mod (x, n) = mod (y, n) IMPLIES
divides (n, x - y)

  mod_eq_mod1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n /
2)) :
    (i /= j) IMPLIES (mod (2 * i + n / 2 - 3, n)) /= (mod (2 * j + n / 2 - 3, n))

  UnrollMod1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i : subrange (1, n / 2)) :
    mod (2 * i + n / 2 - 3, n) =
      IF ((2 * i + n / 2 - 3) < n) THEN (2 * i + n / 2 - 3)
      ELSE ((2 * i + n / 2 - 3) - n)
      ENDIF

  UnrollMod2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i : subrange (n / 2 + 1, n)) :
    mod (2 * (i - n/2) + n / 2 - 3, n) =
      IF ((2 * (i - n/2) + n / 2 - 3) < n) THEN (2 * (i - n/2) + n / 2 - 3)
      ELSE ((2 * (i - n/2) + n / 2 - 3) - n)
      ENDIF

  DiagonalCheck (X1, Y1, X2, Y2 : posnat) : bool =  ((X2 /= X1) AND abs (((Y2 - Y1) / (X2 - X1))) = 1)

  DiagonalCheckSym : LEMMA FORALL (X1, Y1, X2, Y2 : posnat) :
     DiagonalCheck (X1, Y1, X2, Y2) = DiagonalCheck (X2, Y2, X1, Y1)
```

```
%.......................................................................................
%  Intermediate level lemmas used to checking cases 1, 2, and 3 of the algorithm, separately, as they
%  pertain to the "chess" data structure.
%
  ChessMethod1 (n : {t : posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 2)}) : ChessType (n) =
    LAMBDA (i : subrange (1, n)) :
      IF (i <= n / 2) THEN (2 * i)
      ELSE (2 * (i - n / 2) - 1)
      ENDIF

  ChessMethod2 (n : {t : posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}) : ChessType (n) =
    LAMBDA (i : subrange (1, n)) :
      IF (i <= n / 2) THEN (1 + (mod (2 * i + n / 2 - 3, n)))
      ELSE (n - mod (2 * (n + 1 - i) + n / 2 - 3, n))
      ENDIF

  FinalChess (n : posnat | n > 3) : ChessType (n) =
   IF even? (n) AND (mod (n, 6) /= 2) THEN ChessMethod1 (n)
   ELSIF even? (n) THEN ChessMethod2 (n)
   ELSIF (mod (n - 1, 6) /= 2) THEN
     (LAMBDA (j : subrange (1, n)) : IF (j = n) THEN n ELSE ChessMethod1 (n - 1)(j) ENDIF)
   ELSE (LAMBDA (j : subrange (1, n)) : IF (j = n) THEN n ELSE ChessMethod2 (n - 1)(j) ENDIF)
   ENDIF

%.......................................................................................
%  Lemmas for checking conflict along a row/col/diag. Note, these lemmas are independent of the
%  "chess" data structure.
%
%  Lemmas for case 1 and 3 of the algorithm.
%
  M1Perm1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3)}, i, j : subrange (1, n)) :
    (i /= j) IMPLIES (2 * i) /= (2 * j - 1)

  M1Diag1 : LEMMA FORALL (n : {t: posnat | (t > 3)}, i, j : subrange (1, n)) :
    (i /= j) IMPLIES NOT DiagonalCheck (i, 2 * i, j, 2 * j)

  M1Diag2 : LEMMA FORALL (n : {t: posnat | (t > 3)}, i, j : subrange (n / 2 + 1, n)) :
    (i /= j) IMPLIES NOT DiagonalCheck (i, 2 * (i - n / 2) - 1, j, 2 * (j - n / 2) - 1)

  M1Diag3_1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 2)}, i : subrange (1, n / 2), j :
subrange (n / 2 + 1, n)) :
    NOT DiagonalCheck (i, 2 * i, j, 2 * (j - n / 2) - 1)

  M1Diag3_2 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange (1, (n - 1) / 2)) :
    NOT DiagonalCheck (i, 2 * i, ((n - 1) / 2 + i), 2 * i - 1)

  M1Diag3_3 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3) AND (mod (t - 1, 6) /= 2)}, i : subrange (1, (n - 1)
/ 2), j : subrange ((n + 1) / 2, n - 1)) :
    NOT DiagonalCheck (i, 2 * i, j, 2 * (j - (n - 1) / 2) - 1)

  M1Diag3_4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i, j : subrange ((n + 1) / 2, n - 1)) :
    (i /= j) IMPLIES NOT DiagonalCheck (i, 2 * (i - (n - 1) / 2) - 1, j, 2 * (j - (n - 1) / 2) - 1)

  M1Diag4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange (1, (n - 1) / 2)) :
    NOT DiagonalCheck (i, 2 * i, n, n)

  M1Diag5 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange ((n + 1) / 2, n - 1)) :
    NOT DiagonalCheck (i, 2 * (i - (n - 1) / 2) - 1, n, n)
```

```
%
% Lemmas for case 2 and 3 of the algorithm.
%

  M2Perm1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n / 2)) :
    (i /= j) IMPLIES (1 + (mod (2 * i + n / 2 - 3, n))) /= (1 + (mod (2 * j + n / 2 - 3, n)))

  M2Perm2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n / 2)) :
    (1 + (mod (2 * i + n / 2 - 3, n))) /= (n - (mod (2 * j + n / 2 - 3, n)))

  M2Diag1 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n / 2)) :
    (i /= j) IMPLIES NOT DiagonalCheck (i, 1 + (mod (2 * i + n / 2 - 3, n)), j, 1 + (mod (2 * j + n / 2 - 3, n)))

  M2Diag2 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n / 2)) :
    (i /= j) IMPLIES NOT DiagonalCheck (n + 1 - i, n - (mod (2 * i + n / 2 - 3, n)), n + 1 - j, n - (mod (2 * j + n / 2 - 3, n)))

  M2Diag3 : LEMMA FORALL (n : {t: posnat | even? (t) AND (t > 3) AND (mod (t, 6) /= 0)}, i, j : subrange (1, n / 2)) :
    NOT DiagonalCheck (i, 1 + (mod (2 * i + n / 2 - 3, n)), n + 1 - j, n - (mod (2 * j + n / 2 - 3, n)))

  M2Diag4 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3) AND (mod (t - 1, 6) /= 0)}, i : subrange (1, (n - 1) /
2)) :
    NOT DiagonalCheck (i, 1 + (mod (2 * i + (n - 1) / 2 - 3, (n - 1))), n, n)

  M2Diag5 : LEMMA FORALL (n : {t: posnat | odd? (t) AND (t > 3)}, i : subrange ((n + 1) / 2, n - 1)) :
    NOT DiagonalCheck (i, (n - 1) - (mod (2 * (n - i) + (n - 1) / 2 - 3, (n - 1))), n, n)

%.........................................................................................
% The main properties.
%
  QueensSolution (n : posnat, ct : ChessType (n)) : bool =
    FORALL (i, j : subrange (1, n)) :
      (i /= j IMPLIES ct (i) /= ct (j)) AND
      (i /= j IMPLIES NOT DiagonalCheck (i, ct (i), j, ct (j)))

  QueensFinal : THEOREM FORALL (n : {t: posnat | (t > 3)}) :
    QueensSolution (n, FinalChess (n))

  EightQueens : THEOREM FORALL (n : {t : posnat | (t > 3)}, Chess : ChessType (n)) : QueensSolution (8,
FinalChess (8))

%.........................................................................................

END Queens
```

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
1-04-2020	Technical Memorandum	

4. TITLE AND SUBTITLE	**5a. CONTRACT NUMBER**
Formal Verification of a Solution to the n-Queens Problem	**5b. GRANT NUMBER**
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	**5d. PROJECT NUMBER**
Malekpour, Mahyar R..	**5e. TASK NUMBER**
	5f. WORK UNIT NUMBER 340428.02.20.07.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	**8. PERFORMING ORGANIZATION REPORT NUMBER**
NASA Langley Research Center Hampton, VA 23681-2199	L-21067
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)** NASA
National Aeronautics and Space Administration Washington, DC 20546-0001	**11. SPONSOR/MONITOR'S REPORT NUMBER(S)** NASA-TM-2020-220588

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-
Subject Category 62
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes a formal verification of a concise algorithm that computes a solution to the n-Queens problem for all natural numbers n, such that n > 3. The formal proof of the algorithm is completed in the Prototype Verification System (PVS) theorem prover. This verification effort serves two purposes. First, it is presented as a pedagogical example for learning a theorem prover, such as PVS, and second, as a candidate benchmark for comparing other formal methods tools to PVS.

15. SUBJECT TERMS

Formal Proof; PVS; Verification; n-Queens

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	18	**19b. TELEPHONE NUMBER** *(Include area code)* (757) 864-9658