A Bounded Model Checker for SPARK Programs

Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto

HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. This paper discusses the design and implementation of a bounded model checker for SPARK code, and provides a proof of concept of the utility and practicality of bounded verification for SPARK.

Introduction. SPARK is a programming language and toolset designed for the development of high-assurance software [4]. The language is based on a restricted subset of Ada (see [1] for a full description), complemented by an expressive system of contracts, to describe the specification and design of programs. The SPARK platform provides a set of tools that allow users to reason about the correctness of the source code, making possible the detection of problems early in the software lifecycle. The tools are based on deductive verification and as such give full guarantees, but they require the user to provide *contracts* and *loop invariants*, which are often difficult to write.

We believe that *Bounded Model Checking* (BMC) of software can be helpful as a complement to the existing tools, particularly for finding errors and/or validating annotations, or in assisting with the conversion of existing Ada code to SPARK. The key idea of BMC of software, as implemented by the flagship CBMC tool [6] for ANSI-C code, is to encode *bounded behaviors* of the program as logical formulas whose models describe executions leading to violations of some given property. The use of *assert* and *assume* annotations provide a convenient property specification mechanism. An `assert` ϕ statement signifies that the property ϕ should hold at that point of the program. If ϕ does not hold, that violation is reported and a counter-example is given. Asserts can be used, for instance, to automatically instrument the code regarding safety properties. On the other hand, an `assume` ψ annotation states that one can rely on the fact that ψ is true at that point of the program.

This paper presents the design of a BMC tool for SPARK 2005 programs. Rather than producing the definitive BMC tool for SPARK, which would not make sense at the present moment of development of the language (SPARK 2014 had not been launched when our development began), the main goal of the present paper is to provide a proof of concept of the utility and practicality of BMC of SPARK software, as a complement to deductive verification. A bounded model checker may also serve other purposes in addition to formal verification: in [2] it is reported how CBMC has been used for *coverage analysis* of safety-critical code, in precisely the same context in which SPARK is widely used. Other applications of BMC of software include *automated fault localization* [9].

```
package Marray is
 Array_Size: constant:=10;
 subtype Ind is Integer range 1 .. Array_Size;
 type VArray is array (Ind) of Integer;
 procedure MaxArray(V: in VArray; M: out Ind);
 --# derives M from V;
 --# post (for all I in Ind => (V(I) <= V(M)));
end Marray;

package body Marray is
 procedure MaxArray(V: in VArray; M: out Ind) is
  I: Integer;
  Max: Ind;
 begin
  Max := Ind'First;
  --% notOverflow(+,Integer,INDICES'FIRST,1);
  I := Ind'First+1;
  loop
   --# assert(for all J in Ind range Ind'First..(I-1) => (V(J) <= V(Max)));
   --# assert(I >= Ind'First) and (I <= Ind'Last + 1);
   exit when I > Ind'Last;
   --% assert (I >= VARRAY'FIRST) and (I <= VARRAY'LAST);
   --% assert (MAX >= VARRAY'FIRST) and (MAX <= VARRAY'LAST);
   if V(I) > V(Max) then
     Max := I;
   end if;
   --% notOverflow(+,Integer,I,1);
   I := I + 1;
  end loop;
  M := Max;
 end MaxArray;
end MArray;
```

SPARK-BMC. SPARK-BMC[1] is an open source prototype bounded model checker for SPARK programs. It is developed in Haskell and uses as backend the SMT solver Z3 [8]. The tool checks SPARK programs for violations of properties annotated in the code. Annotations are inserted as comments beginning with a user predefined character, assumed distinct from SPARK annotations (`--%` in this paper). The annotations that may be used are `assert C`; `assume C`; and `notOverflow(op,type,e1,e2)`, used to check if an overflow is originated. These can be inserted to express useful properties for debugging, and in particular safety properties corresponding to the absence of runtime exceptions (such as overflow, array out-of-bounds accesses and division by zero), which can be checked without requiring invariants. The figure shows a SPARK program (including loop invariants) containing a procedure that finds the maximum element in an array. The annotations corresponding to overflow and array out of bounds, to be checked by SPARK-BMC, are also included.

The reader is referred to [13] for a complete description of the implementation details that will now be outlined. The algorithm begins with a normalization into a subset of SPARK (e.g. transforming type attributes and enumerations into integer expressions) and inlining of routine calls. SPARK does not allow for any form of recursion, so no bound is applied on the length of this expansion. Loops are then unwound a fixed number of times (which reduces them to sequences of nested `if` statements), and the program is thus transformed into a monolithic iteration-free program. To enforce soundness, an unwinding assertion can be optionally inserted, to ensure that the loop has been sufficiently unwound.

[1] Available from the repository `https://bitbucket.org/vhaslab/spark-src`

In order to extract a logical formula from the iteration-free program one has to first transform it into a *single assignment* form, in which the values of the variables do not change once they have been used (so that assignments can be seen as logical equalities). The program is then subject to Conditional Normal Form (CNF) normalization, which transforms it into a sequence of statements of the form `if b then S`, where `S` is an assignment, assert or assume statement, and the guard b encodes the path condition leading to that command. Two sets of formulas $\mathcal{C}$ and $\mathcal{P}$ are then extracted. $\mathcal{C}$ describes logically the operational contents of the program, and includes a formula $b \rightarrow x = e$ for every statement `if` b `then` $x := e$. $\mathcal{P}$ on the other hand contains the properties to be established, extracted from the guarded assert and assume statements of the CNF. If no assert fails in any execution of the program one has that $\bigwedge \mathcal{P}$ is a logical consequence of $\mathcal{C}$. Any model found for the set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ corresponds to a counter-example: an execution leading to a violation of some assertion in $\mathcal{P}$.

Experiences with implementing SMT-based bounded model checkers [3,7] have produced quite positive results, which justifies our choice of this class of solvers. SPARK-BMC employs a bit-vector rather than an unbounded integers theory, which has the advantage of capturing precisely the low-level fixed-width machine semantics of program data types. The SPARK programming language includes modular types, whose modular semantics are directly captured by fixed-size bit-vectors. Signed integers are also conveniently encoded as bit-vectors. Finally, arrays are modeled by a theory of arrays.

Evaluation. At the present stage of development of SPARK-BMC we can successfully check multi-procedure programs manipulating arrays and discrete types, and as such we are able to run SPARK-BMC on a great variety of programs. Our preliminary results clearly illustrate the positive aspects of automated verification for SPARK code. The tool scales well for certain classes of programs, and even for other, algorithmically more complicated programs, it is able to check for property violations in the first few iterations of loops. Let us turn back to the `MaxArray` example to show how the tool can discover subtle bugs without the need for user annotations. One common error would be to write the `exit` condition as `exit when I>Ind'Last+1`, which would cause an array out of bounds exception in the array access contained in the expression `V(I)>V(MAX)` that could easily be missed. The SPARK tools (based on deductive verification) would generate a Verification Condition (VC) stating that the loop invariant is preserved by iterations of the loop, and another VC to enforce that whenever `V(I)>V(MAX)` is evaluated the value of `I` lies within the range of the array. For the `MaxArray` code both VCs are successfully discharged, but if the `exit` condition is modified to the above, then the invariant preservation condition can no longer be proved (it fails in the last iteration). If the invariant is corrected to `I<=Ind'Last+2`, then the invariant preservation VC is discharged, but not the other VC: the invariant is now correct, but does not prevent the out-of-bounds access. This illustrates that with deductive verification it can be hard to detect exactly what went wrong – is the program unsafe, or is the user-provided invariant wrong? To use SPARK-BMC on this program, it must first be annotated as discussed before. Depending on the user-provided bound K, SPARK-BMC will either indicate that the unwinding assertion fails, or else (for $K > 10$) that an assert violation occurs. In this case the tool displays the violated assertion, as well as the current values of the relevant variables.

We have assessed the behaviour of SPARK-BMC with a number of example programs, taken both from academic papers and from problem sets proposed in the context of program verification competitions. We have either transcribed to SPARK an algorithm implemented in C, or, when this was not available, coded it from scratch. All the code can be found in the project's repository.

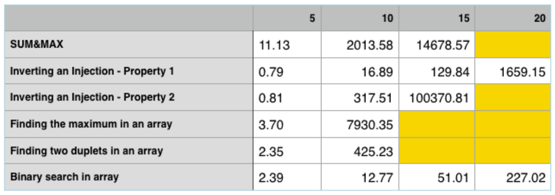
Experimental Results: Problem Set I. We use a first set of example programs to illustrate that BMC can be used in practice on programs that have been designed to be verified with deductive verification tools. Although these are all relatively simple problems, they are algorithmically complicated, which creates difficulties for a BMC approach. Our purpose here is to investigate the viability of the approach for small problem sizes (bounded loops requiring up to 100 iterations). Specifically, we have applied SPARK-BMC to an implementation of the *inverting an injection* problem taken from [12], which we have ported to SPARK:

```
MAXLEN: constant := 20;

subtype Index
        is Integer range 0..MAXLEN;
type ArrayType
      is array (Index) of Integer;

procedure Invert(A: in ArrayType;
                 N: in Integer;
                 B: out ArrayType) is
begin
 for I in Index range 0 .. N - 1
 loop
  B(A(I)) := I;
 end loop;
end Invert;
```

```
procedure PropertyCheck is
 A,B: ArrayType;
 N: Integer;
begin
 --% assume (N > 0 and N <= MAXLEN);
 for I in Index range 0 .. N - 1 loop
  --% assume (A(I) >= 0 and A(I) < N);
  for J in Index range I + 1 .. N - 1 loop
   --% assume (A(I) /= A(J));
  end loop;
 end loop;
 Invert(A,N,B);
 for I in Index range 0 .. N - 1 loop
  for J in Index range I + 1 .. N - 1 loop
   --% assert (A(I) /= A(J));
  end loop;
 end loop;
 for I in Index range 0 .. N - 1 loop
  --% assert (B(A(I)) = I);
 end loop;
end PropertyCheck;
```

The problem is described as follows: *Invert an injective (and thus surjective) array A of N elements in the subrange from 0 to N-1.* The verification tasks are to prove that the output array `B` is injective and that `B(A(I)) = I for 0 <= I < N`. SPARK-BMC succeeds in both tasks for the given bound, with no further annotations in addition to the assumes and asserts shown in the `PropertyCheck` procedure, which state that the properties described above hold after calls to `Invert`, for executions corresponding to an injective array `A`.

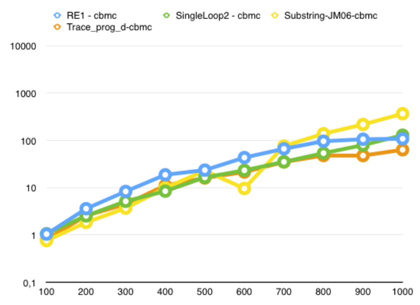
We additionally tested the tool with the following: *SUM&MAX* from [12]; *finding the maximum in an array* and *finding two duplets in an array* from [5]; and finally *binary search in an array* from [14]. The table below shows the verification time (in seconds) vs. the number of iterations unwound, as required by the problem size. While in all cases, for a sufficiently small number of iterations, SPARK-BMC succeeds in the verification task in a completely automatic way, it easily becomes impractical to reach even a modest number of iterations.

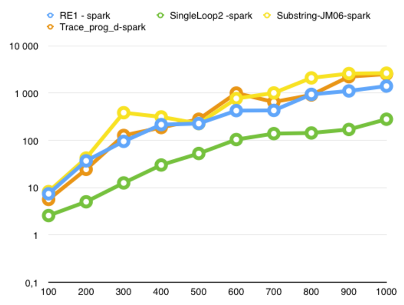


	5	10	15	20
SUM&MAX	11.13	2013.58	14678.57	🟨
Inverting an Injection - Property 1	0.79	16.89	129.84	1659.15
Inverting an Injection - Property 2	0.81	317.51	100370.81	🟨
Finding the maximum in an array	3.70	7930.35	🟨	🟨
Finding two duplets in an array	2.35	425.23	🟨	🟨
Binary search in array	2.39	12.77	51.01	227.02

Experimental Results: Problem Set II. We use both SPARK-BMC and CBMC on a second set of example programs (running on the SPARK and C versions of the same algorithm). In order to compare both tools at a purely logical level, the times registered for CBMC were measured with the *constant propagation and simplification* option switched off. We stress that our goal with these comparisons is not to present SPARK-BMC as competing with CBMC – we aim merely to validate the algorithm underlying SPARK-BMC, and demonstrate the practicality of bounded verification with a diverse set of problems.

The programs in this second set have been used to illustrate the performance of various software model checking and symbolic execution tools [10,11][2]. Although we do not present results obtained with these tools, they would surely outperform BMC tools, since the example programs are designed to illustrate situations that are advantageous to them. The programs are algorithmically simpler than in the previous set, and would be straightforward to verify deductively. Bounded verification scales quite well for these programs, with reasonable verification times for up to 1000 iterations. The graphs below show that in all programs CBMC (even with constant propagation switched off) performs better than SPARK-BMC. However, it can be seen that in a log-lin scale the shapes of the SPARK-BMC curves are relatively close to the CBMC curves.





In fact, when we run CBMC on the examples from the first problem set (times not shown in the graph), SPARK-BMC behaves marginally better than CBMC with *binary search in array*, and much better than CBMC with *inverting an injection* (CBMC becomes impractical to use with just 15 iterations). It seems that for algorithmically more complicated code the size of the propositional formulas generated by CBMC increases very significantly.

[2] The C code can be found online at `http://map.uniroma2.it/smc/simp/` and `http://www.cfdvs.iitb.ac.in/~bhargav/dagger.php`

Conclusion. We have demonstrated the advantages of bounded SMT-based automated verification for SPARK code, and shown that it is practical to check for property violations with no loop invariant annotations required. In the near future our work will focus on adapting SPARK-BMC to work with the SPARK 2014 language definition, as well as on including support for other SMT solvers. An interesting challenge, which will certainly increase the usefulness of SPARK-BMC as a complement to the SPARK tools, is to extend it in order to validate and debug SPARK contracts.

Acknowledgment. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project **FCOMP-01-0124-FEDER-020486**.

References

1. Altran Praxis. *SPARK - The SPADE Ada Kernel (including RavenSPARK)*, December 2011, Edition 7.2.
2. D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Autom. Reason.*, 45(4):397–414, Dec. 2010.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, Jan. 2009.
4. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 2003.
5. T. Bormer, et al. The cost IC0701 verification competition 2011. In *FoVeOOS*, pages 3–21, 2011.
6. E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*. Springer, 2004.
7. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In ASE '09, IEEE Computer Society, 2009.
8. L. de Moura and N. Bjørner. *Z3: An Efficient SMT Solver*, volume 4963 of *LNCS*. Springer, 2008.
9. A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science*, 174(4):95 – 111, 2007.
10. J. Jaffar, J. Navas, and A. Santosa. Unbounded symbolic execution for program verification. In *Runtime Verification*, volume 7186 of *LNCS*. Springer, 2012.
11. R. Jhala and K. McMillan. A practical and complete approach to predicate refinement. In *TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
12. V. Klebanov, et al. The 1st verified software competition: Experience report. In *FM 2011*, volume 6664 of *LNCS*. Springer, 2011.
13. C. B. Lourenço. A Bounded Model Checker for SPARK Programs. Master's thesis, University of Minho, 2013.
14. B. Weide, et al. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, Experiments*, volume 5295 of *LNCS*. Springer, 2008.