SPARK-BMC: Checking SPARK Code for Bugs

Cláudio Belo Lourenço, Victor Cacciari Miraldo, Maria João Frade, and Jorge Sousa Pinto

HASLab/INESC TEC & Universidade do Minho, Portugal

Abstract. The standard SPARK deductive verification tools, based on contracts, are not practical in early stages when the idea is only bug catching. We discuss the implementation of a bounded model checker for SPARK, focusing on specific challenges of this language. Our tool is fully automatic, complementing the existing tools for SPARK.

1 Introduction

Program verification techniques can be grouped in two main families. While *deductive verification* based on the use of a program logic and the design-by-contract principle gives full guarantees and allows for expressing properties using a rich behavior specification language, it is not automatic: it is the user's responsibility to provide *contracts* and other information required for verification to proceed, such as *loop invariants*. Such information requires a lot of effort from the user since it is often difficult to write. The second family of techniques is based on model checking, more precisely *model checking of software*, which typically allows only for simpler properties, expressed as assertions in the code, but is fully automated. The fundamental idea is to create a model from the source program, and then, given a property, check if it holds in that model. However, such an approach has a main downside: state space explosion. This problem is common to all applications of model checking, but the presence of data makes it worse in the case of software.

Bounded Model Checking (BMC) [1] is an approach that can be used to overcome this limitation by only checking execution paths with size up to a fixed (user-provided) bound, sacrificing soundness. The general idea is that only a partial exploration of the state space is performed. Even so, BMC is useful as long as it finds bugs that would otherwise be missed.

SPARK is a programming language and tool set designed for the development of high-assurance software [2]. The SPARK language is a subset of Ada, complemented by an expressive system of contracts (inserted as Ada comments) to describe the specification and design of programs. The existing tools[1] for verification of SPARK programs are mainly based on deductive verification, which implies that someone has to write the contracts and loop invariants that will be used to generate verification conditions, which is in general far from straightforward. To fill this gap, we propose here an automated verification tool for SPARK

[1] Mainly commercialized by Altan Praxis, `http://www.altran-praxis.com`

code: a bounded model checker inspired by the success of the CBMC tool for bounded model checking of C programs [3].

2 Background

BMC of software. The key idea of BMC of software is to encode *bounded behaviors* of the program that enjoy some given property as a logical formula whose models, if any, describe execution paths leading to a violation of the property. The properties to be established are assertions on the program state, included in the program through the use of *assert statements*. For every execution of the program, whenever a statement `assert` ϕ is met, the assertion ϕ must be satisfied by the current state, otherwise we say that the state violates the assertion ϕ. The verification technique assumes that a satisfiability-based tool is used to find models corresponding to property violations.

At the heart of a BMC tool stands an algorithm that extracts a logical formula directly from the source code (including properties expressed as assertions), without user intervention. The algorithm begins with some preliminary simplification steps that may include the removal of side effects, or normalization into a subset of the target programming language. The next step is where information is lost, in the sense that only bounded behaviors are preserved. Given an entry-point provided by the user, the program is expanded by unwinding loops a fixed number of times, and inlining routine calls (in the presence of recursion, a bound is also applied on the length of this expansion). A program consisting of multiple routines is thus transformed into a monolithic program, which is both recursion-free and iteration-free. To enforce soundness of BMC, an *unwinding assertion* can be placed at the end of each expanded segment of code. If the unwinding assertion (negating the condition of the loop) is not violated by any execution, then checking the transformed (bounded) code is sound. Unwinding assertions can be omitted, in which case one must always bear in mind the unsoundness of the approach. In this case the unwinding `assert` ϕ statement is replaced by `assume` ϕ, to ignore execution paths requiring more iterations.

In order to extract a logical formula from this monolithic program, we have to transform it into a form in which the values of the variables do not change once they have been used (so that they can be seen as logical variables). This is done by converting the program into a *single-assignment* (SA) form in which multiple indexed versions of each variable are used – a new version is introduced for each assignment to the original variable, so that in every execution path, once a variable has been read or assigned it will not be assigned again. The next step is to normalize this program into *conditional normal form* (CNF): a sequence of single-branch conditional statements of the form `if` b `then` S, where S is an *atomic statement*, i.e. either an assignment, *assert*, or *assume* instruction. The idea is to flatten the branching structure of the program, so that every atomic statement is guarded by the path condition leading to it.

At this point, two sets $\mathcal{C}$ and $\mathcal{P}$ can be extracted from the program: $\mathcal{C}$ includes a formula $b \rightarrow x = e$ for every statement `if` b `then` $x := e$ and a formula $b \rightarrow \theta$

```
package Marray is
   Array_Size: constant:=10;
   subtype Indices is Integer range 1 .. Array_Size;
   type VArray is array (Indices) of Integer;

   procedure MaxArray(V: in VArray; M: out Indices);
   --# derives M from V;
   --# post (for all I in Indices => (V(I) <= V(M)));
end Marray;
```

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

Fig. 1. SPARK program specification and body: the procedure receives an array of integer values and returns the index of the maximum element.

for every statement `if` b `then assume` θ; $\mathcal{P}$ includes a formula $b \rightarrow \phi$ for every statement `if` b `then assert` ϕ. $\mathcal{C}$ captures logically the operational contents of the program, and $\mathcal{P}$ contains the properties to be established.

If no assert statement fails in any execution of the program one has that $\bigwedge \mathcal{P}$ is a logical consequence of $\mathcal{C}$. This can be determined by checking the satisfiability of the set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$. Any model found for it corresponds to an execution path that leads to an assertion violation. Of course, satisfiability checking is restricted to models that capture the properties of the data structures manipulated by the program, and that are specified by some background theory $\mathcal{T}$ (usually a combination of several theories). Therefore 'satisfiability' should in fact be understood as $\mathcal{T}$-satisfiability. It can be checked by a Satisfiability (SAT) or an Satisfiability Modulo Theory (SMT) solver: the main difference is the way numeric values and arrays are modelled.

SPARK. Although SPARK [2] is based on a heavily restricted subset of Ada together with a set of annotations, it should be considered in its own right as a full language for the development of annotated high-assurance software. Annotations in SPARK code (lines starting with `--#`) must be written as comments, ignored by compilers but not by the SPARK verification tools.

A SPARK program is composed by one or more units. There exist two different kinds of program units: *packages* and *subprograms*. Subprograms define computations, which may have parameters with mode `in`, `out` or `in out` and are divided into *functions* and *procedures*. Functions are routines with only mode `in` parameters, that may not have side effects, and always return a value. Each function must have exactly one *return* statement (the last statement in the function). Function calls occur inside expressions, while procedure calls occur as standalone statements (procedure may not contain return statements). Package, is used as a way of grouping related entities (e.g. data types and objects, subprograms or even nested packages). All program units are generally divided in two parts: specification (the program unit's interface) and body (the implementation details). Fig. 1 contains the package `Marray`'s specification and body.

Many features of Ada are not present in SPARK because they are considered 'dangerous' or difficult to verify in the development of safety-critical systems. Some of these exclusions facilitate our work in the development of a BMC for SPARK, for instance there is no need to limit the number of times a function is inlined (recursion is not allowed); the same applies to pointer validity checks.

SPARK is not just a language, but also a set of tools to check if a program respects all the restrictions imposed on valid SPARK programs, and also for program verification. The Examiner is a tool responsible for performing syntactic and static semantic analyses for checking the validity of SPARK programs, as well as generating verification conditions. The reader is referred to [2] for a full description of the SPARK tools, as well as SPARK restrictions.

Types are organized into categories [2]. For instance, the *discrete* type hierarchy is devided into *integer* types and *enumerations*. Moreover, integer types are devided into *signed* and *modular* types. Operations over the former may result in overflow (which raise runtime exceptions), whereas operations over the later have wraparound semantics. A modular type is defined by giving a power of two integer N; its values range from 0 to $N-1$. In the example, `T` is a modular type.

```
type T is mod 4;
type Nat is range 0 .. Integer'Last;
type Day is (Mon,Tue,Wed,Thur,Fri,Sat,Sun);
subtype Counter is Nat range 1 .. 10;
subtype Weekday is Day range Day'First .. Fri;
```

The range of the predefined integer type is defined in a default SPARK package, but it can also be given in a configuration file. It is also possible to define new integer types, with range given by two static expressions (lower and upper bound), as for example as `Nat` shown above. Note the use of the expression `Integer'Last` which makes use of an attribute over the integer type. An enumeration is defined using a list of identifiers (enumeration literals). In SPARK, as opposed to Ada, enumeration literals cannot be overloaded. Type `Day` is an example of a declared enumeration type. Comparison operators may be applied to enumeration types, except for type `Boolean`, where the only operators are equality and inequality.

A *subtype* of a certain type is defined by giving a lower and upper bound over the base type. One can define subtypes of both enumeration types and

integer types, but not of modular types. For instance, `Counter` and `Weekday` are examples of subtypes of the types `Nat` and `Day` respectively.

In addition to discrete types there exist also *composite types*, divided into *records* and *arrays*. A record is a structure consisting of named components. As in many other languages, an array consists of an indexed list of elements of the same type. The index must be a discrete type and may possibly be *constrained*. `Tuple` in the example below, is an unconstrained array (note the use of `<>`) while `WorkHours` is a constrained array. Objects of an array type must always have a static bound, therefore the types `Tuple` and `Matrix` cannot be used directly to create new objects, since they have unconstrained indexes.

```
type Tuple is array (Integer range <>) of Day;
type WorkHours is array (Weekday) of Nat;
type Matrix is array (Counter, Integer range <>) of Circle;
```

Turning now to SPARK statements, the most primitive form of iteration is implemented by an infinite `loop` control structure and an explicit abrupt `exit` command which should always occur under an if statement without else branch, or alternatively within a `when` clause. An exit statement always refers to the innermost loop. Every `while` or `for` loop can be converted into a primitive loop.

3 SPARK-BMC

SPARK-BMC is a prototype bounded model checker for SPARK programs, that follows closely the BMC algorithm described before. The tool checks *valid* SPARK programs for property violations as will be explained below. It assumes, without checking, that the input program passes the Examiner validity checks. The tool is being developed in Haskell and uses as backend the SMT-solver Z3 [4]. SPARK-BMC is open source; it is available from the repository `https://bitbucket.org/vhaslab/spark-src`.

The tool parses the SPARK program and creates an Abstract Syntax Tree (AST) with the program representation. The program is then transformed, simplified, and normalized (by multiple traversals of the AST) and two sets of logical formulas are extracted, describing logically the operational contents of the program, and the properties to be established. Finally, the SMT solver is used to check if there exists a property violation; if so, a counter-example is shown.

The tool checks for properties annotated in the code which are inserted as comments beginning with `--%`, distinct from SPARK annotations. A program annotated for SPARK-BMC is still a valid SPARK program that can be checked by the usual SPARK toolset. Our tool works with the following annotations:

```
--% assert C;
```

```
--% assume C;
```

```
--% notOverflow(op,type,e1,e2);
```

The two basic annotations `assert C` and `assume C`, with `C` a quantifier-free formula, are similar to those used in CBMC: `assert C` means that the assertion `C` must be satisfied by the current state, and `assume C` means that all paths violating the property are ignored. The `notOverflow` is used to check if overflow

occurs and is translated into an assert with a formula that depends on the model used for numeric values (bit-vectors or unbounded integers).

```
--% notOverflow(+,ITER,INDICES'FIRST,1);
I := Indices'First+1;
...
--% assert (I >= VARRAY'FIRST(1)) and (I <= VARRAY'LAST(1));
--% assert (MAX >= VARRAY'FIRST(1)) and (MAX <= VARRAY'LAST(1));
if (V(I) > V(MAX)) then
...
--% notOverflow(+,ITER,I,1);
I := I + 1;
```

Fig. 2. Annotated program

Similarly to CBMC, these annotations can be inserted by the user to express properties that should hold (very helpful for instance for debugging purposes), or else added automatically by an instrumentation tool that analyses the program and inserts 'obvious' annotations. In SPARK code it is particularly useful to check statically for runtime exceptions; the SPARK Examiner certainly does this, but it requires annotating loop invariants in the code. These properties (in particular overflow, array out of bound access and division by zero) can be instrumented automatically, and with SPARK-BMC they can be checked without requiring invariants, as will be illustrated later in the paper. For overflow, the instrumentation inserts a `notOverflow` annotation for each arithmetic operation that can possibly cause overflow; for division by zero it inserts an `assert` to check that the denominator is different from zero; for array out of bounds access, it inserts an `assert` to check the validity of each index. Fig. 2 shows some properties that may be automatically inserted in the example of Fig. 1.

The rest of this section presents the details of the several transformation steps on the original program, and the interaction with the SMT solver.

Simplification. The first step is to rewrite the input program into an equivalent one that uses only a restricted set of statements, namely, `loop`, `exit`, `if-then`, `if-then-else` and assignment. `case` statements are translated into `if` statements. `exit-when` statements are rewritten using `if` and `exit` statements. Also `if` statements using the `elsif` keyword are translated into nested `if`s. `while` and `for` loops are rewritten as discussed using `loop`, `if` and `exit` statements.

Subprogram inlining. The inlining of routine calls consists of taking the entry point provided by the user and recursively remove subprogram calls. For subprograms used as standalone statements, this is done by simply replacing the subprogram call by the respective body. For function calls occurring as part of expressions, the function body is inserted exactly before the statement which contains the call, and an auxiliary variable is used to propagate the return value. Auxiliary variables are also used to propagate the values of parameters inside the callee and back to the caller subprogram, taking into account their modes

(`in`, `out` or `in out`). Since different contexts are being merged, identifiers are renamed to avoid conflicts, by adding as prefix the package and subprogram name (this is also useful to keep information about the identifier's context). The result of this transformation step is a monolithic program with no calls.

Eliminating attributes and enumeration literals. SPARK attributes apply to types and subtypes. Since we assume that the program being verified is always a valid SPARK program, it is known beforehand that the enumeration literals are being used correctly. Moreover, SPARK forbids the overloading of enumeration literals. With this in mind and the information about declarations obtained in the last step, we translate enumeration literals into integer values (the respective position in the enumeration) and get rid of attributes, replacing them by equivalent expressions. The only exception is for literals of Boolean type, since they can be used as conditions and the Boolean type is not ordered.

Loop unwinding. Loops can in general be seen as blocks of code containing backward-goto and forward-goto statements. Since the iteration schemes have been removed, only primitive loops are present at this stage, and the only way of exiting such a loop is through an explicit exit, equivalent to a forward-goto statement. On the other hand, reaching the end of a loop block produces a backward-goto to the beginning of the loop.

In the present step, in order to produce a bounded model, each loop gets unwound a fixed number of times K. Loop headers are removed, and loop bodies are replicated K times. However, exit statements complicate this picture: the statements belonging to the loop body following a reached exit statement do not get executed. This is done by introducing an artificial loop unwind wrapper for the code standing between the exit statement and end of the unwound loop code. With this wrapper, reaching an exit statement means that none of the following statements inside the wrapper should get executed. Such wrappers have other uses as will be seen below, but will be removed in the final normalization step. An unwinding assertion or assumption (depending on the user choice) is placed at the end of the expanded code to produce the effect explained in Section 2.

Single-assignment transformation. A crucial step towards the normalization of the program is its transformation into an SA form. In this transformation, multiple indexed versions of each variable are used – a new version is introduced for each assignment to the original variable, so that in every execution path, once a variable has been read or assigned it will not be assigned again.

While the transformation of straight-line code is quite straightforward, code with multiple branches poses some challenges. The only statement left with multiple branches at this stage of the transformation workflow is the `if` statement (exit statements inside loop wrappers will be discussed later). The two final versions of each variable (one from each branch) must be merged; this is achieved by inserting, immediately after the conditional, an assignment to each modified variable, making use of *conditional expressions*, as used in C.

Our conversion algorithm traverses the AST and appends an appropriate index to each variable occurrence. To deal with multiple branch statements, the algorithm makes use of two counters for each variable; one family of counters (`R`) keeps the index that should be used when a variable is read, the other (`W`) keeps the last index of the variables that have been used for writing. Both `R(v)` and `W(v)` are incremented when variable `v` is assigned. However, when entering an *else* branch `R(v)` must be reset to the value it had immediately before the entering of the `if` conditional. At the merge point, the values of `W` at the end of both branches are used for inserting an assignment with the appropriate conditional expression. Consider the example

```
                                        -- R = {(X,2), (Y,4)}; W = {(X,3),(Y,4)}
if (X#2 = Y#4) then
  X#4 := X#2 + Y#4;                     -- R = {(X,4), (Y,4)}; W = {(X,4),(Y,4)}
else
  X#5 := X#2 + 5;                       -- R = {(X,5), (Y,4)}; W = {(X,5),(Y,4)}
end if;
X#6 := (X#2 = Y#4) ? X#4 : X#5;
                                        -- R = {(X,6), (Y,4)}; W = {(X,6),(Y,4)}
```

The fact that `R(X)` and `W(X)` are different indicates that this is part of some *else* branch. Observe how the value of `R(X)` before entering the conditional is used in both branches when `X` is read. Special attention must be given to assignments involving arrays. For the purpose of obtaining a logical encoding, arrays are seen as *aplicative*: each array correspond to a single variable, updated through a *store* function. So an assignment of the form `A(X) := Y` is first transformed into `A := store(A,X,Y)`, and conversion to SA form then produces the instruction `A2 := store(A1,X',Y')`, where `X'` and `Y'` correspond to `X` and `Y` with possible renaming of variables due to transformation to SA form. Fig. 3 presents the state of our running example after conversion to SA form. Note in particular that `MAX#7` is not assigned in this iteration because its valued depends on the reached `exit` statement. Such assignments are added in the next iteration when removing `exit` statements.

CNF normalization. The program is normalized into a sequence of statements of the form `if b then S`, where `S` must be an *assignment*, *assert* or *assume* instruction. To reach such form, `exit` statements must be removed. This step is divided in two parts: first, the program is normalized in the form described above, allowing `S` to be an `exit` statement; later this statement is removed.

For the first part, `if` statements with `else` branch are rewritten into a sequence of two `if` statements with mutually exclusive conditions. This transformation is correct since in SA form the value of the Boolean expression cannot possibly be modified by subsequent instructions. Nested `if` statements are then pushed up in the AST, by traversing it and collecting the necessary path conditions for reaching each assignment, exit, assert or assume instruction, and then creating an `if` statement with the conjunction of the collected conditions.

For the exit statements removal, recall that when such a statement is reached, none of the following instructions in the loop wrapper should be reached. Moreover, the values of the variables after the wrapper must be in accordance with the

```
ARRAY_SIZE#1 := 10;
MAX#2 := 1;
--% notOverflow(+,INTEGER,1,1);
I#2 := (1 + 1);
LOOP_WRAPPER
  if (I#2 > ARRAY_SIZE#1) then
    exit;                                   -- [(I,2),(M,1),(MAX,2),(V,1)]
  end if;
  --% assert (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1);
  --% assert (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1);
  if (V#1(I#2) > V#1(MAX#2)) then
    MAX#3 := I#2;
  end if;
  MAX#4 := (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2;
  --% notOverflow(+,INTEGER,I#2,1);
  I#3 := (I#2 + 1);
  if (I#3 > ARRAY_SIZE#1) then
    exit;                                   -- [(I,3),(M,1),(MAX,4),(V,1)]
  end if;
  --% assert (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1);
  --% assert (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1);
  if (V#1(I#3) > V#1(MAX#4)) then
    MAX#5 := I#3;
  end if;
  MAX#6 := (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4;
  --% notOverflow(+,INTEGER,I#3,1);
  I#4 := (I#3 + 1);
  --% assert False;
END_LOOP_WRAPPER;                           -- [(I,5),(MAX,7)]
M#2 := MAX#7;
```

Fig. 3. Single-assignment program representation

first exit statement reached. If no exit statement is reached then an `assert` or an `assume` annotation must be reached. The preparation for this step starts when the code is converted into SA form: for each exit statement, the current version of each variable at that point is kept, and at the end of each loop wrapper, a new version for each modified variable is reserved. This version is the one that is used immediately after the loop wrapper: its value reflects the exit statement that has been reached. This information is displayed as comments in Fig. 3.

At this stage each exit statement is guarded by a condition `C`; the statement is reached if and only if `C` evaluates to true. To ensure that none of the subsequent statements (including other exits) are reached after an exit statement is, one has to ensure that none of the subsequent guards evaluate to true, by propagating the condition `not C` through them. In order for variables to hold the correct values after the wrapper, assignments to the reserved variables are inserted using the current variables that were kept together with the exit statement. The guard for these assignments is the exit guard. Loop wrappers may now be removed. Fig. 4 shows the CNF representation of our running example.

Creating the logical encoding. There are no language-specific aspects in the next step. After all the previous transformations, the program is now a sequence of statements of the form `if C then S`, where `S` may only be an assignment or an instruction derived from an annotation (`assert`, `notOverflow`, or `assume`). As described in Section 2, one now has to extract two sets of formulas $\mathcal{C}$ and

```
-- ...
True => _NExit#1 := not((I#2 > ARRAY_SIZE#1));
(I#2 > ARRAY_SIZE#1) => MAX#7 := MAX#2;
(I#2 > ARRAY_SIZE#1) => I#5 := I#2;
_NExit#1 => --% assert (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1);
_NExit#1 => --% assert (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1);
_NExit#1 and (V#1(I#2) > V#1(MAX#2)) => MAX#3 := I#2;
_NExit#1 => MAX#4 := (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2;
_NExit#1 => --% notOverflow(+,INTEGER,I#2,1);
_NExit#1 => I#3 := (I#2 + 1);
True => _NExit#2 := not((I#3 > ARRAY_SIZE#1)) and _NExit#1;
_NExit#1 and (I#3 > ARRAY_SIZE#1) => MAX#7 := MAX#4;
_NExit#1 and (I#3 > ARRAY_SIZE#1) => I#5 := I#3;
_NExit#2 => --% assert (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1);
_NExit#2 => --% assert (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1);
_NExit#2 and (V#1(I#3) > V#1(MAX#4)) => MAX#5 := I#3;
_NExit#2 => MAX#6 := (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4;
_NExit#2 => --% notOverflow(+,INTEGER,I#3,1);
_NExit#2 => I#4 := (I#3 + 1);
_NExit#2 => --% assert False;
True => M#2 := MAX#7;
```

Fig. 4. CNF normalization (auxiliary variable `_NExit` may be propagated)

$\mathcal{P}$. This is straightforward, by traversing the list of statements, transforming `if` statements into implications and assignments into equalities. The formulas with assert statements are collected in $\mathcal{P}$, and the remaining implications in $\mathcal{C}$.

Solver interaction. Now that we have $\mathcal{C}$ and $\mathcal{P}$ the satisfiability of $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ modulo a background theory has to be checked. The simultaneous presence in the language of discrete types with modular semantics and of signed integers for which runtime overflow exceptions are raised led us to elect fixed-size bit-vectors for our primary encoding of discrete types: the modular semantics is directly captured by bit-vectors, and for signed integers, since overflow is protected by instrumented `notOverflow` annotations, it is indifferent to use bit-vectors or an unbounded integers encoding. Our choice was to use Z3 [4] as proof tool. Z3 is a high-performance SMT solver being developed at Microsoft Research. Open source bindings[2] are available for Haskell, which allow for the use of some convenient features, as well as a direct interaction with the solver.

The first task is to create the adequate Z3 sorts for each SPARK predefined or user-defined type. A relation between these two classes must be kept for building Z3 expressions. One may then create one Z3 constant for each program variable. Since Z3 does not support multiple-index arrays, SPARK arrays with multiple indexes are represented using nested arrays in Z3. Formulas from $\mathcal{C}$ are interpreted and sent to the solver one by one, while formulas from $\mathcal{P}$ are used to build the expression $\neg \bigwedge \mathcal{P}$ which is then also sent to Z3. The interpretation of SPARK expressions and creation of Z3 formulas is programmatic, using functions available in the Z3 bindings. The Z3 bindings also provide a function for checking satisfiability, and in the SAT case produces a model to be used as counter-example.

[2] `http://hackage.haskell.org/package/z3`

4 Conclusion

Although the development of SPARK-BMC is still in progress (some features of SPARK are not yet covered), the tool's workflow is entirely implemented, and we are able to successfully check programs manipulating arrays and discrete types. Our preliminary results are satisfactory, and sufficient to illustrate the advantages of automatic verification. Let us turn back to the example in Fig. 1 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 follows:

```
exit when I > Indices'Last + 1;
```

It is easy to see that this alternative condition would cause an array out of bounds exception in the array access contained in the expression `V(I) > V(MAX)`, but such a bug can be easily missed. Our tool detects it automatically.

The SPARK tools (based on deductive verification) would generate a verification condition (labelled as `assert`) stating that the loop invariant is preserved by iterations of the loop, and another VC (labelled as `rtc check`) to enforce that whenever `V(I) > V(MAX)` is evaluated the value of `I` lies within the range of the array. For the code of Fig. 1 both VCs are successfully discharged: no out-of-bounds access takes place. 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). The `rtc check` is still proved, because it is a consequence of the invariant. If the invariant is corrected to `I <= Indices'Last + 2`, then the invariant preservation VC is discharged, but not the `rtc check` – the invariant is now correct, but it does not prevent the out-of-bounds access. This example 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, the program is first automatically instrumented as discussed in Section 3. Depending on the bound K provided by the user, 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 variables occurring in them. Safety violations like overflow and division by zero would also be detected if present.

Even though not particularly optimized at the present stage, the tool seems to scale reasonably well (the array size in the example may well be increased up to thousands). Our current work focuses on covering aspects of SPARK that are still not handled, including support for floating and fixed point types and respective properties. In the near future we will work on optimizing the generation of Z3 expressions for scalability. At a later stage we intend to (i) extend the tool to handle concurrent programs, more precisely programs written in the RavenSPARK profile, by employing context-bounded analysis; and (ii) investigate how the tool can be used for coverage analysis of SPARK code, following work in this direction (for C code) using CBMC [5].

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. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149
2. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2003)
3. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In Jensen, K., Podelski, A., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 168–176
4. de Moura, L., Bjørner, N. In: Z3: An Efficient SMT Solver. Volume 4963/2008 of Lecture Notes in Computer Science. Springer Berlin (April 2008) 337–340
5. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. J. Autom. Reason. **45**(4) (December 2010) 397–414