Program Verification in SPARK and ACSL: A Comparative Case Study

Eduardo Brito and Jorge Sousa Pinto
{edbrito,jsp}@di.uminho.pt

Departamento de Informática / CCTC
Universidade do Minho, Braga, Portugal

Abstract. We present a case-study of developing a simple software module using contracts, and rigorously verifying it for safety and functional correctness using two very different programming languages, that share the fact that both are extensively used in safety-critical development: SPARK and C/ACSL. This case-study, together with other investigations not detailed here, allows us to establish a comparison in terms of specification effort and degree of automation obtained with each toolset.

1 Introduction

In recent years, deductive program verification based on contracts and JML-like annotation languages has been a very active and fruitful area of research. The state of the art of currently available tools has been greatly advanced with the use of SMT provers and other automatic proof tools targeted for verification.

The SPARK [1] programming language and toolset offers program verification capabilities as part of a wider array of static analyses aimed at the development of high-integrity software. The SPARK reality is different from that faced by verification tools for general-purpose languages, since the SPARK language itself is so restricted (as imposed by the specific application domains in which it is used) that some of the big verification challenges (to name one: the manipulation of data structures in the program heap) are not even present.

Curiously, our industrial research partners who work in the safety-critical context (more specifically, in the development of real-time, embedded applications) are mainly interested in two programming languages: SPARK and C. This may sound surprising, as one can hardly think of two imperative languages that stand farther away from each other in terms of safety restrictions, but it is a reality. Prompted by this fact, we present in this paper an attempt to compare SPARK with C in terms of the programming and annotation languages, as well as the currently available verification tools. Naturally, this comparison only makes sense for a subset of C that excludes features that are absent in SPARK.

Let us recall two fundamental differences between both programming languages, in addition to the safety issues: C has very little support for abstraction, whereas SPARK, which is a subset of Ada, explicitly supports abstract data types, as well as refinement. In SPARK specification contracts are part of the language; in C we resort to the external ACSL [3] specification language.

- `nat count()` – Returns the number of elements currently in the stack.
- `nat capacity()` – Returns the maximum number of elements that the stack may contain.
- `boolean isEmpty()` – Returns information on whether the stack is empty.
  Postcond: `Result = (count() = 0)`
- `boolean isFull()` – Returns information on whether the stack is full.
  Postcond: `Result = (count() = capacity())`
- `int top()` – Returns the top of the stack.
  Precond: `not isEmpty()`
- `void pop()` – Removes the top of the stack.
  Precond: `not isEmpty()`; Postcond: `count() = old_count() - 1`
- `void push(int n)` – Pushes item n onto the stack.
  Precond: `not isFull()`; Postcond: `count() = old_count() + 1` and `top() = n`

Fig. 1. Stack operations

Our goals are (i) to explain the differences involved in coding a very simple software module when full verification is an issue; (ii) to evaluate the relative difficulty of establishing the safe execution of programs in both platforms (we expect it to be easier in SPARK, as a consequence of the language design); and (iii) to assess how the verification tools compare in terms of automatic proof.

The paper can be used by readers familiar with either ACSL (or other JML-like language) or SPARK, as a quick introduction to the other platform, and as a general introduction to verified development in both languages. We believe this to be a useful contribution towards promoting the use of such tools.

2 Background

The goal of deductive program verification is to *statically* ensure that a program performs according to some intended specification, resorting to the axiomatic semantics of the programming languages and tools like theorem provers. Typically, what is meant by this is that the input/output behaviour of the implementation matches that of the specification (this is usually called the *functional* behaviour of the program), and moreover the program does not 'go wrong', for instance no errors occur during evaluation of expressions (the so-called *safety* behaviour). Related approaches that do not concern us here are *dynamic verification* (which considers a single run of a program), *software model checking* (based on the exploration of a limited state space), and *extended static checking* (which abandons correctness and completeness for the sake of automation). Neither of these offer as high an assurance degree as deductive verification.

The idea of a software *contract* – consisting, for each procedure / method, of a precondition that should be established by the caller and a postcondition that must be established by the callee – was initially meant to be used as part of a software development cycle that relies on dynamic verification. The code is compiled by a special compiler that introduces run-time checks for the contracts in the code, that will be executed at call-time and at return-time. Since these conditions are checked dynamically, they must be written as boolean expressions in the syntax of the programming language, which may include calls to other methods or functions defined as part of the same program.

Figure 1 shows a typical specification of a bounded stack that can be found in many tutorials on the design by contract approach to software development [13]. The figure contains an informal description of each operation on stacks, and in some cases a contract consisting of a precondition and a postcondition. Notice that methods `count`, `capacity`, and `isFull` occur in several preconditions and postconditions. In fact, the first two are not usually given as part of a stack's interface, and their presence is justified by their use in other methods' contracts.

In general, annotation languages include two features that can be found in postconditions in this example: the possibility of referring to the value of an expression in the pre-state (`old_count()` for `count`), and of referring to the return value (`Result`). The preconditions state that some stack operations cannot be performed on an empty or a full stack, while the postconditions partially specify the functional behaviour of the methods. This is straightforward for `isEmpty` and `isFull`. For `push` the postcondition ensures that the element at the top of the stack on exit is indeed the pushed value, and the stack count is increased with respect to its initial value; for `top` the contract simply states that the count is decreased. It is implicit that the stack remains unmodified, with the exception of its top element when performing a push or pop operation.

Although program verification based on preconditions and postconditions predates design by contract by decades, it has been revitalized by the growing popularity of the latter and the advent of specification languages like JML [10], intended to be used by different tools ranging from dynamic checking to test-case generation, static analysis and verification. In contract-based program verification each procedure C is annotated with a contract (Precond: P; Postcond: Q); checking its correctness amounts to establishing the validity of the Hoare triple $\{P\}\, C\, \{Q\}$ [8]. A program is correct if all its constituent annotated procedures are correct. The verification process follows the mutually recursive nature of programs: in proving the correctness of procedure f that invokes procedure g, one simply assumes the correctness of g with respect to its contract. In a deductive framework, correctness of a program can be established by the following steps.

1. Annotating the source code with specifications in the form of contracts (for every procedure / function / method) and invariant information for loops;
2. Generating from the code, with the help of a *verification conditions generator* tool (VCGen for short), a set of first-order proof obligations (verification conditions, VCs), whose validity will imply the correctness of the code; and
3. Discharging the verification conditions using a proof tool. If all VCs are valid then the program is correct.

ACSL/Frama-C and SPARK. Frama-C [3] is a tool for the static analysis of `C` programs. It is based on the intermediate language Jessie [11] and the multi-prover VCGen Why [7]. `C` programs are annotated using the ANSI-C Specification Language (ACSL). Frama-C contains the `gwhy` graphical front-end that allows to monitor individual verification conditions. This is particularly useful when combined with the possibility of exporting the conditions to various proof tools, which allows users to first try discharging conditions with one or more

automatic provers, leaving the harder conditions to be studied with the help of an interactive proof assistant. For the examples in this paper we have used the `Simplify` [6] and `Alt-Ergo` [5] automatic theorem provers. Both Frama-C and ACSL are work in progress; we have used the Lithium release of Frama-C.

SPARK on the other hand is both a language and a toolset. The language is a strict subset of the Ada 95 standard, with some added anotations, designed with predictability and safety in mind. What we mean by strict is that every SPARK program is a valid Ada 95 program. This is very important since the SPARK toolset does not provide a compiler, relying instead on existing Ada compilers. A clearly defined semantics for the SPARK subset of Ada 95 is obtained by imposing a set of rules that precisely define a set of programming practices and limitations that do not depend on specific aspects of the compiler

Because SPARK was created mainly to be used in the context of critical embedded and real-time systems, it imposes some restrictions that may seem too harsh, but are in fact fairly standard in those scenarios. For instance in embedded systems it is usually important to know the exact memory footprint of the programs, so dynamic memory allocation is forbidden in SPARK. Also, pointers / pointer operations and recursion are not present in SPARK.

The most relevant tools in the toolset are the Examiner, the Simplifier, and the interactive Proof Checker. The Examiner is responsible for checking if the Ada code is compliant with the set of restrictions imposed by SPARK, including the consistency of programs with respect to data and information flow annotations. It also contains the VCGen functionality responsible for generating the proof obligations. The Simplifier tool simplifies and attempts to automatically discharge the verification conditions, with the help of user-supplied rules. Although not a powerful automatic theorem prover of the same nature as those used by Frama-C, it is a carefully designed tool that incorporates many years of experience in simplifying typical VCs. The Proof Checker is the manual, interactive prover. For this work we have used SPARK GPL edition 2009, V. 8.1.1.

3 Bounded Stack: Specification

We use the bounded stack example to illustrate the differences between the verified development of a small software module in SPARK and C/ACSL. We first discuss a few modifications of the typical DbC specification in Figure 1. If we think algebraically in terms of the usual stack equations:

$$top(push(n, s)) = n \qquad\qquad pop(push(n, s)) = s$$

Only the first equation is ensured by the contracts of Figure 1. Note that the contracts for `push` and `pop` do not state that the methods preserve all the elements in the stack apart from the top element; they simply specify how the *number* of elements is modified. We will strengthen the specification by introducing a *substack* predicate, to express the fact that a stack is a substack of another. The notion of substack together with the variation in size allows for a complete specification of the behaviour of these operations. Equally, the contracts for `top`,

```
package Stack
--# own State: StackType;
is
   --# type StackType is abstract;
   --# function Count_of(S: StackType) return Natural;
   --# function Cap_of(S: StackType) return Natural;
   --# function Substack(S1: StackType; S2: StackType) return Boolean;

   MaxStackSize: constant := 100;

   procedure Init;
   --# global out State;
   --# derives State from;
   --# post Cap_of(State) = MaxStackSize and Count_of(State) = 0;

   function isEmpty return Boolean;
   --# global State;
   --# return Count_of(State) = 0;

   function isFull return Boolean;
   --# global State;
   --# return Count_of(State) = Cap_of(State);

   function Top return Integer;
   --# global State;
   --# pre Count_of(State) > 0;

   procedure Pop;
   --# global in out State;
   --# derives State from State;
   --# pre 0 < Count_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)-1 and
   --#      Substack(State, State~);

   procedure Push(X: in Integer);
   --# global in out State;
   --# derives State from State, X;
   --# pre Count_of(State) < Cap_of(State);
   --# post Cap_of(State) = Cap_of(State~) and Count_of(State) = Count_of(State~)+1 and
   --#      Top(State) = X and Substack(State~, State);
end Stack;
```

Fig. 2. Stack SPARK specification

`isEmpty`, and `isFull` must state that these methods do not modify the stack (i.e. they have no side effects, which is not stated in Figure 1).

Additionally, we add to the specification an initialisation function that creates an empty stack. Also, we consider that the operations `count` and `capacity` are not part of the interface of the data type (they are not available to the programmer). In both specification languages `count` and `capacity` will be turned into the *logical functions* `count_of` and `cap_of` that exist only at the level of annotations, and are not part of the program. These logical functions are sometimes also called *hybrid functions* because they read program data. In ACSL they are declared inside an `axiomatic` section at the beginning of the file. Note that no definition or axioms can be given at this stage for the logical functions.

In SPARK (as in Ada) the specification and implementation of a package are usually placed in two separate files: the *package specification* (`.ads`) and the *package body* (`.adb`) containing the implementation. Packages are separately

compiled program units that may contain both data and code and provide encapsulation. Figure 2 shows the specification file for the Stack package; `StackType` is an abstract type that is used at the specification level and will later be instantiated in a package body. In the package specification a variable `State` of type `StackType` stands for an abstract stack, i.e. an element of the ADT specified. This will be refined in the body into one or more variables of concrete types.

The specification of a bounded stack in ACSL is given in Figure 3. For the sake of simplicity we choose to use a global stack variable, but stacks could equally be passed by reference to the C functions. A crucial difference with respect to the SPARK specification is that ACSL has *no support for refinement* (and neither has C, of course). Thus in the figure the `typedef` declaration is left unfinished. The reader should bear in mind that it will not be possible to reason about stacks without first providing a concrete implementation. Whereas in SPARK/Ada one can have different implementations in different body files for the same package specification file, in C those implementations would have to be obtained using the file in the figure as a *template* that would be expanded.

Some language features are directly reflected in the two specifications. The SPARK function `Init` will always produce an empty stack with capacity given by the constant `MaxStackSize`, since dynamic allocation is not possible. In the C version it takes the desired stack capacity as argument. Also, we take advantage of SPARK's type system and set the type returned by functions `Cap_of` and `Count_of` to `Natural` rather than `Integer` (since the number of elements cannot be negative). C's type system is much less precise, thus integers are used instead, but note the use of the `integer` ACSL logical type (for logical functions only).

Concerning the two specification languages, different keywords are used to identify preconditions (`pre`, `requires`) and postconditions (`post`, `ensures`), as well as the return values (`return`, `\result`). Also, ACSL offers the possibility of using optional *behaviours* in specifications, which permits the association of more than one contract to a function. For instance the behaviour `empty` (resp. `not_empty`) of function `isEmpty` corresponds to the precondition that the current count is zero (resp. not zero), specified with an *assumes* clause. Behaviours allow for more readable specifications and for more structured sets of VCs.

C functions may in general have side effects, whereas SPARK functions are by definition *pure*: they are not allowed to modify the global state or to take parameters passed by reference. Thus the SPARK functions `isEmpty`, `isFull`, and `top` are not allowed to modify the state of the stack, which is an improvement (obtained for free) with respect to the contracts in Figure 1. In ACSL functions can be annotated with *frame conditions* that specify the modified parts of the state (variables, structure fields, array elements, etc). The frame conditions of the above three pure functions are written `assigns \nothing`. Appropriate verification conditions are generated to ensure the validity of each frame condition.

A consequence of the previous difference is that SPARK allows for program functions to be used in assertions, whereas in ACSL this is forbidden because of the possibility of side effects. This is reflected in different treatments of the `Top` function in both languages: in the SPARK specification `Top` is a program function

```
typedef ... Stack;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = ...
  @ logic integer top_of{L} (Stack st) = ...
  @ logic integer count_of{L} (Stack st) = ...
  @ predicate substack{L1,L2} (Stack st) = ...
  @ } */

/*@ requires cap >= 0;
  @ ensures cap_of{Here}(st) == cap  && count_of{Here}(st) == 0;
  @*/
void init (int cap);

/*@ assigns \nothing;
  @ behavior empty:
  @    assumes count_of{Here}(st) == 0;
  @    ensures \result == 1;
  @ behavior not_empty:
  @    assumes count_of{Here}(st) != 0;
  @    ensures \result == 0;
  @*/
int isEmpty (void);

/*@ assigns \nothing;
  @ behavior full:
  @    assumes count_of{Here}(st) == cap_of{Here}(st);
  @    ensures \result == 1;
  @ behavior not_full:
  @    assumes count_of{Here}(st) != cap_of{Here}(st);
  @    ensures \result == 0;
  @*/
int isFull (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures \result == top_of{Here}(st);
  @ assigns \nothing;
  @*/
int top (void);

/*@ requires 0 < count_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) - 1 &&
  @         substack{Here,Old}(st);
  @*/
void pop(void);

/*@ requires count_of{Here}(st) < cap_of{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) &&
  @         count_of{Here}(st) == count_of{Old}(st) + 1 &&
  @         top_of{Here}(st) == x && substack{Old,Here}(st);
  @*/
void push (int x);
```

Fig. 3. Stack ACSL specification: operation contracts

```
with Stack;
--# inherit Stack;
package SSwap is
  procedure Swap(X, Y: in out Integer);
   --# global in out Stack.State;
   --# derives Stack.State, X, Y from Stack.State, X, Y;
   --# pre Stack.Count_of(Stack.State) <= Stack.Cap_of(Stack.State)-2;
   --# post X = Y~ and Y = X~;
end SSwap;

package body SSwap is
  procedure Swap(X, Y: in out Integer)
   is
   begin
      Stack.Push(X); Stack.Push(Y);
      X := Stack.Top; Stack.Pop;
      Y := Stack.Top; Stack.Pop;
   end Swap;
end SSwap;
```

Fig. 4. Swap using a stack

and it is used in the postcondition of `Push`, whereas in ACSL a new logical function `top_of` is used; its relation with the `top` program function is established by a postcondition of the latter. In addition to logical / hybrid functions, ACSL offers the possibility of having *predicates* to be used in annotations; they may be either defined or else declared and their behaviour described by means of axioms. In SPARK a predicate must be declared as a logical function that returns a boolean. This is reflected in the declarations of `substack` in both languages.

In ACSL it is possible to refer to the value of an expression in a given program state, which is extremely useful in any language with some form of indirect memory access. In fact, all hybrid functions and predicates *must* take as extra arguments a set of *state labels* in which the value of the parameters are read, even if this set is singular. Thus, for instance, whereas in SPARK the `substack` predicate takes two stacks as arguments, and is invoked (in the postconditions of `Pop` and `Push`) with arguments `State` and `State~`, where the latter refers to the state of the stack in the pre-state, the ACSL version takes as arguments a single stack variable *st* and two state labels L_1, L_2 , with the meaning that the value of *st* in state L_1 is a substack of the value of *st* in state L_2. It is then invoked in annotations making use of predefined program labels *Here* (the current state) and *Old* (the pre-state in which the function was invoked).

In SPARK the procedures `Init`, `Pop`, and `Push` have data flow annotations with the meaning that the state of the stack is both read and modified, and the new state depends on the previous state (and for `Push` also on the argument `X`). In functions, the `--# global State;` data flow annotation simply means that these functions read the state of the stack. At this abstract level of development, it is not possible to specify with either SPARK data flow annotations or ACSL frame conditions that the procedures do not modify some *part* of the state (e.g. `pop` and `push` preserve the capacity). This has then to be done using postconditions.

Reasoning about Specifications in SPARK. A major difference between both languages is that in SPARK it is possible to reason in the absence of concrete implementations. To illustrate this, we will define a procedure that swaps the values of two variables using a stack. The relevant package and body are shown in Figure 4. Running the SPARK Examiner on this file produces 9 verification conditions, of which, after running the SPARK Simplifier, only one is left unproved. This VC is generated from the postcondition of `Swap`, which is only natural since we haven't given a definition of `substack`.

The SPARK Simplifier allows users to supply additional rules and axioms, in the FDL logical language, in a separate file. The following SPARK rule states that two equally sized substacks of the same stack have the same top elements.

```
ss_rule(1) : stack__top(S1) = stack__top(S2) may_be_deduced_from
  [stack__count_of(S1) = stack__count_of(S2), stack__substack(S1,S3), stack__substack(S2,S3)].
```

Unfortunately, even with this rule, the Simplifier fails to automatically discharge the VC, so the user would be forced to go into interactive proof mode (using the SPARK Proof Checker) to finish verifying the program. Alternatively, the following rule allows the Simplifier to finish the proof automatically:

```
ss_rule(3) : stack__top(S1) = stack__top(S2) may_be_deduced_from
  [stack__count_of(S3) = stack__count_of(S2)+1, stack__count_of(S1) = stack__count_of(S3)-1,
   stack__substack(S1,S3), stack__substack(S2,S3)].
```

This also illustrates a technique that we find very useful with the Simplifier: writing special purpose rules that follow the structure of the computation. In this example we have simply mentioned explicitly the intermediate stack S_3 that the state goes through betwen S_2 and S_1. This is often sufficient to allow the Simplifier to discharge all VCs without the need for interactive proof.

4 Bounded Stack: Implementation / Refinement

Figure 5 shows a fragment of the stack package implementation, including the definition of the state and the definition of the `Push` procedure. The corresponding fragment in C is given in Figure 6. The state is in both cases defined as a set of two integer variables (for the size and capacity) together with an array variable. In SPARK a *range type* `Ptrs` is used, which is not possible in C.

In C we simply fill in the template of Figure 3 without touching the annotations. We consider a straightforward implementation of bounded stacks as structures containing fields for the capacity and size, as well as a dynamically allocated array. This requires providing, in addition to the C function definitions, appropriate definitions of the logical functions `cap_of`, `top_of`, and `count_of`, as well as of the predicate `substack`. `count_of` and `cap_of` simply return the values of structure fields. The most sophisticated aspect is the use of a universal quantifier in the definition of `substack`. Note also the use of the operator `\at` to refer to the value of a field of a structure variable in a given program state (not required when a single state label is in scope – it is implicit).

```
package body Stack
--# own State is Capacity, Ptr, Vector;
is
   type Ptrs is range 0..MaxStackSize;
   subtype Indexes is Ptrs range 1..Ptrs'Last;
   type Vectors is array (Indexes) of Integer;

   Capacity: Ptrs := 0;
   Ptr: Ptrs := 0;
   Vector: Vectors := Vectors'(Indexes => 0);

   procedure Push(X: in Integer)
   --# global in out Vector, Ptr;
   --#        in Capacity;
   --# derives Ptr from Ptr & Vector from Vector, Ptr, X & null from Capacity;
   --# pre Ptr < Capacity;
   --# post Ptr = Ptr~ + 1 and Vector = Vector~[Ptr => X];
   is
   begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
      --# accept F, 30, Capacity, "Only used in contract";
   end Push;
```

```
stack_rule(1) : cap_of(S) may_be_replaced_by fld_capacity(S) .
stack_rule(2) : count_of(S) may_be_replaced_by fld_ptr(S) .
stack_rule(3) : count_of(X) = count_of(Y) - Z may_be_replaced_by fld_ptr(Y) = fld_ptr(X) + Z.
stack_rule(4) : count_of(X) = count_of(Y) + Z may_be_replaced_by fld_ptr(X) = fld_ptr(Y) + Z.
stack_rule(5) : count_of(S) = cap_of(S) may_be_replaced_by fld_ptr(S) = fld_capacity(S).
stack_rule(6) : substack(X, Y) may_be_deduced_from
  [V=fld_vector(X), Z=fld_ptr(X)+1, Z=fld_ptr(Y), fld_vector(Y)=update(V, [Z], N)].
stack_rule(7) : substack(X, Y) may_be_deduced_from
  [fld_vector(X)=fld_vector(Y), fld_ptr(X)<fld_ptr(Y)].
stack_rule(8) : stack__top(X) = Y may_be_deduced_from
  [fld_vector(X) = update(Z, [fld_ptr(X)], Y)] .
```

Fig. 5. Stack SPARK implementation (fragment) and user-provided rules

SPARK on the other hand has explicit support for refinement. Thus contracts can be written at a lower level using the state variables, as exemplified by the `Push` procedure. Since there are no logical definitions as such in SPARK, the functions `cap_of` and `count_of` will be handled by the user rules `stack_rule(1)` and `stack_rule(2)` that can be applied as rewrite rules in both hypotheses and conclusions. The user rules 3 to 5 are auxiliary rules; their presence illustrates the limitations of the Simplifier in applying the previous 2 rewrite rules.

Refinement Verification in SPARK. Invoking the SPARK examiner with both package and body files will produce a set of verification conditions, establishing a correspondence between specification and implementation contracts in the classic sense of refinement: given a procedure with specification precondition P_s (resp. postcondition Q_s) and body precondition P_b (resp. postcondition Q_b), the VCs $P_s \Longrightarrow P_b$ and $Q_b \Longrightarrow Q_s$ will be generated, together with conditions for correctness of the procedure's body with respect to the specification (P_b, Q_b).

A crucial refinement aspect of our example has to do with the `substack` predicate. Note that there is no mention of the predicate at the implementation level, so we must now provide rules for inferring when a stack is a substack of

```
typedef struct stack {
  int capacity;
  int size;
  int *elems;
} Stack;

int x, y;
Stack st;

/*@ axiomatic Pilha {
  @ logic integer cap_of{L} (Stack st) = st.capacity;
  @ logic integer top_of{L} (Stack st) = st.elems[st.size-1];
  @ logic integer count_of{L} (Stack st) = st.size;
  @ predicate substack{L1,L2} (Stack st) = \at(st.size,L1) <= \at(st.size,L2) &&
  @  \forall integer i; 0<=i<\at(st.size,L1) ==> \at(st.elems[i],L1) == \at(st.elems[i],L2);
  @ predicate stinv{L}(Stack st) =
  @  \valid_range(st.elems,0,st.capacity-1) && 0 <= count_of{L}(st) <= cap_of{L}(st);
  @ } */

/*@ requires count_of{Here}(st) < cap_of{Here}(st) && stinv{Here}(st);
  @ ensures cap_of{Here}(st) == cap_of{Old}(st) && count_of{Here}(st) == count_of{Old}(st)+1
  @  && top_of{Here}(st) == x && substack{Old,Here}(st) && stinv{Here}(st);
  @*/
void push (int x) {
  st.elems[st.size] = x;
  st.size++;
}

/*@ ensures x == \old(y) && y == \old(x);
  @*/
swap() {
  init(3); push(x);  push(y); x = top(); pop(); y = top(); pop();
}
```

Fig. 6. Stack C implementation (extract) and test function (`swap`)

another. Writing a rule based on the use of a quantifier (as we did in ACSL) would not help the Simplifier (although it could be used for interactive proof), thus we provide instead rule (6) for the specific case when X is a substack of Y that contains only one more element (`fld_vector` and `fld_ptr` correspond to the fields `Vector` and `Ptr` respectively in the stack body), and rule (7) regarding the case of two stacks represented by the same vector with different counters. These basically describe what happens in the `push` and `pop` operations.

In these rules we make use of the fact that SPARK arrays are logically modeled using the standard theory of arrays [14], accessed through the `element` and `update` operations. In particular the expression `update(V, [Z], N)` denotes the array that results from array `V` by setting the contents of the position with index `Z` to be `N`. Rule (8) concerns the top of a stack after an update operation at the `ptr` position. With these rules the Simplifier is able to discharge all VCs.

Verification of C code. Our C/ACSL file now contains a full implementation of the stack operations, based on the previously given contracts. Let us add to this a `swap` function (also shown in Figure 6). Running Frama-C on this file will generate verification conditions that together assert that the code of the stack operations and of the `swap` function conforms to their respective contracts. 38

VCs are generated, only 4 of which, labelled "pointer dereferencing", are not discharged automatically. These are safety conditions, discussed below.

Safety Checking. Being able to write exception-free code is a very desirable feature in embedded and critical systems. In the stack example this is relevant for array out-of-bounds access, and again the two languages offer different approaches. An important feature of SPARK is that, using proof annotations and automatically generated safety conditions, programs can be shown statically not to cause runtime exceptions. The expression *runtime checks* (or *safety conditions*) designates VCs whose validity ensures the absence of runtime errors.

In the SPARK implementation the domain type of the array is a range type (as are the other state variables), which in itself precludes out-of-bounds access. The runtime errors that may occur concern precisely the range types: every use of an integer expression (in particular in assignments and array accesses) will generate conditions regarding the lower and upper bounds of the expression. For instance the instruction `Ptr := Ptr + 1` in the `Push` procedure generates a VC to check that `ptr + 1` lies within the range of type `Indexes`. Such conditions are generated and automatically discharged in both the `swap` and the refinement verification in a completely transparent way.

ACSL on the other hand treats array accesses (and pointer dereferencing in general) through special-purpose annotations. This is motivated by the very different nature of arrays in C – in particular they can be dynamically allocated and no range information is contained in their types. A `valid_range` annotation in a function precondition expresses that it is safe for the function to access an array in a given range of indexes. It should also be mentioned that a memory region separation assumption is used by default when reasoning about arrays.

Frama-C automatically introduces verification conditions for checking against out-of-bound accesses, thus the 4 VCs left unproved in our example. In order to address this issue we create a new predicate `stinv` that expresses a safety invariant on stacks (the count must not surpass the capacity, and array accesses should be valid within the range corresponding to the capacity). It suffices to include this predicate as precondition and postcondition in all operation contracts (with the exception of the precondition of `init`) for the safety conditions to be automatically discharged. The modifications are already reflected in Figure 6.

5 Conclusion

We are of course comparing two very different toolsets, one for a language with dynamic memory and 'loose' compilation, and another for a memory-bounded language with very strict compilation rules and side-effects explicitly identified in annotations (not to mention the refinement aspect). From our experience with SPARK and the study of published case studies the Simplifier does a very good job of automatically discharging safety conditions. The Simplifier has been compared with SMT solvers, and the relative advantages of each discussed [9].

While it would be unfair to compare SPARK with Frama-C in terms of the performance of safety checking (in particular because SPARK benefits from the

strict rules provided by Ada regarding runtime exceptions), we simply state that safety-checking ACSL specifications requires an additional effort to provide specific safety annotations, whereas in SPARK runtime checks are transparently performed. On the other hand a general advantage of Frama-C is the multi-prover aspect of the VCGen: one can effortlessly export VCs to different provers, including tools as diverse as SMT solvers and the Coq [12] proof assistant. Finally, it is important to remark that unlike SPARK, to this date Frama-C has not, to the best of our knowledge, been used in large-scale industrial projects.

The situation changes significantly when other functional aspects are considered. Take this example from the Tokeneer project, a biometric secure system implemented in SPARK and certified according to the Common Criteria higher levels of assurance (`http://www.adacore.com/tokeneer`). We were quite surprised to find that the Simplifier is unable to prove `C1` from `H20`:

```
H20: element(logfileentries__1, [currentlogfile]) =
     element(logfileentries, [currentlogfile]) + 1 .
-> C1: element(logfileentries, [currentlogfile]) -
       element(logfileentries__1, [currentlogfile]) = - 1 .
```

Simple as it is, our case study has shown that the Simplifier's ability for reasoning with logical functions and user-provided rules is quite limited. Also, our experiences with more 'algorithmic' examples involving loop invariants show that Frama-C is quite impressive in this aspect. For instance fairly complex sorting algorithms, involving nested loops and assertions with quantification, can be checked in Frama-C in a completely automatic manner, with no additional user-provided axioms or rules. In this respect it is our feeling that the SPARK technology needs to be updated or complemented with additional tools.

To sum up our findings, the effort that goes into verifying safe runtime execution is smaller in SPARK, whereas the situation seems to be reversed when the specification and automatic verification of other functional aspects is considered.

One aspect that our running example has not illustrated is related to *aliasing*. Reasoning about procedures with parameters passed by reference is typically difficult because such a procedure may access the same variable through different lvalues, for instance a procedure may access a global variable both directly and through a parameter. In SPARK such situations are rejected by the Examiner after data-flow analysis, so verification conditions are not even generated.

In C such programs are of course considered valid, but note that these situations can only be created by using pointer parameters, and it is possible to reason about such functions with pointer-level assertions. For instance, a function that takes two pointer variables may have to be annotated with an additional precondition stating that the values of the pointer parameters (not the dereferenced values) are different. We have stressed the importance of the use of state labels in ACSL; for reasoning about dynamic structures, serious users of Frama-C will also want to understand in detail the memory model underlying ACSL and the associated separation assumptions, which is out of our scope here.

Finally, we should mention that other tools are available for checking C code, such as VCC [4]. Many other verification tools exist for object-oriented languages; Spec# [2] is a good example.

Acknowledgment. This work was supported by project RESCUE, funded by FCT (PTDC/EIA/65862/2006).

References

1. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
2. Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. In Bertrand Meyer and Jim Woodcock, editors, *Proceedings of VSTTE'05*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.
3. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*. CEA and INRIA. Preliminary design (version 1.4, October 29, 2008).
4. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of TPHOLs'09*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
5. Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: a Theorem Prover for Polymorphic First-order Logic Modulo Theories, 2006. LRI, Univ. Paris-Sud/CNRS, and INRIA.
6. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, 2005.
7. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *Proceedings of CAV'07*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
8. C. A. R. Hoare. An Axiomatic Basis For Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
9. Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT Solvers to Verify High-integrity Programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 60–68, New York, NY, USA, 2007. ACM.
10. Gary T. Leavens. Tutorial on JML, the Java Modeling Language. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Proceedings of ASE'07*, page 573. ACM, 2007.
11. Claude Marché. Jessie: an Intermediate Language for Java and C Verification. In Aaron Stump and Hongwei Xi, editors, *Proceedings of PLPV'07*. ACM, 2007.
12. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2008. Version 8.2.
13. Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10), 1992.
14. John C. Reynolds. Reasoning about arrays. *Commun. ACM*, 22(5):290–299, 1979.