# A verification condition visualizer

# A Verification Condition Visualizer

Madiha Jami and Andrew Ireland

School of Mathematical & Computer Sciences,
Heriot-Watt University,
Edinburgh, EH14 4AS, UK.

**Abstract.** When first encountering data structures such as arrays, records and pointers programmers are often presented with pictorial representations. The use of pictures to describe data structures and their manipulation can help establish basic programming intuitions. The same is true of program proving where pictures are frequently used within the literature to describe program properties such as loop invariants. Here we report on an experimental prototype of a visualization tool that translates verification conditions arising from array based code into pictures. While initially aimed at supporting teaching, we have received positive feedback from users of program proving tools within industry.

## 1 Introduction

The manifesto of the Verified Software Initiative [9] set out a fifteen year programme of research with the aim of demonstrating the viability of formal verification technologies in the development large-scale bug-free software systems. Central in this endeavor are the complementary strands of theory, tools and experiments. Here we focus on *tools* and the need for tools that increase the accessible formal verification techniques. Specifically we are interested in tools that support the teaching of assertion based program proving techniques and which will help win the hearts-and-minds of the next generation of formal methods researchers and practitioners.

While the basic notion of program proof via verification condition generation (VCG) is relatively simple for a toy programming language [4], the approach quickly becomes much harder to teach when working with an industrial-scale programming language and applications. Our language of choice is SPARK[1] [1], a programming language derived from Ada and which is supported by a range of static analysis techniques including formal verification. SPARK has been used extensively within the development of high-integrity software systems, including safety-critical applications such as railway signaling and avionics as well as security-critical application such as smartcard technologies. We have found that the high-profile nature of its applications makes SPARK relatively easy to motivate and is attractive to students. However, when introducing program proving, students find it hard to relate to verification conditions (VCs). Our aim is to use

[1] The version based upon Ada 95.

pictures where appropriate to help programmers gain insight as to the validity of VCs.

As a starting point we have focused on VCs arising from code that manipulates arrays. Whether learning how to construct algorithms that manipulate arrays [13, 3] or how to the reason about the correction of such algorithms [8, 6, 12], authors typically use pictures in order to initially engage their readers. John Reynolds' use of so called *partition diagrams* [12] for reasoning about array based programs is may be the best example. And to a degree it is Reynolds' vision of "making program logics intelligible" that motivates our work.

Here we present an experimental tool that dynamically generates pictures from SPARK VCs. We believe that the generated pictures serve three purposes:

- A picture can more immediately help to identify whether or not a VC is provable.
- If provable then a picture may give guidance as to how a proof of the VC might proceed.
- If a VC is unprovable then the picture may give guidance as to where the bug lies.

While we have emphasized the role of pictures as an aid to teaching, we believe that the power of pictures is more general. For instance, within an industrial context verification engineers will be called upon to deal with the VCs that are not automatically discharged by the proof tools. Deciding whether or not a VC is provable can be a time consuming process, may be even involve wasted interactive proof attempts. If by turning the undischarged VCs into pictures such decisions can be made more quickly then the productivity gains could be significant.

In section 2 we provide a brief overview of the SPARK programming language. Our overall approach is motivated in section 3 while section 4 describes our experimental tool. Related and future work is described in 5 with our conclusions in section 6.

## 2 Background on SPARK

As mentioned above, the focus of our initial experiments has been the visualization of VCs arising from SPARK programs that manipulate arrays. Here we give a brief introduction to the structure of SPARK VCs in general and how arrays are handled specifically. For a more complete description the reader is directed to [1]. SPARK includes an annotation language that supports flow analysis and formal proof. In the case of formal proof the annotations capture the program specification, asserting properties that must be true at particular program points. The annotations are supplied within regular Ada comments, allowing a SPARK compliant program to be compiled using any Ada compiler. Within the work presented here we focus on three proof annotations, namely preconditions (`--# pre`), postconditions (`--# post`) and loop invariants (`--# assert`). When specifying properties of array based programs quantification is important.

SPARK supports both universal (for all) and existential (for some) quantification.

Compliance to the SPARK language is enforced by a static analyser called the Examiner. In addition, the Examiner performs data flow and information flow analysis [2]. The Examiner supports formal verification by building directly upon the Floyd/Hoare style of reasoning. VCs can be generated for proofs of both partial correctness and exception freedom. In the conventional way, arrays are modelled as functions in the programming logic of SPARK, where:

- accessing the $I^{th}$ element of array $A$ is denoted by *element(A, [I])*, while
- updating the $I^{th}$ element with the value $V$ is denoted by *update(A, [I], V)*.

## 3 Our basic approach

Our pictures of array related VCs are based upon boxes for individual elements and rectangles containing ellipses for arbitrary sequences of elements, which we will refer to as *segments*. In terms of referencing elements, we place indexes above the array pictures while properties and relations are depicted using braces below. By way of illustration, Figure 1 gives two pictures. The upper picture shows an array A where all the elements from $f$ to $i-1$ are strictly less than the $i^{th}$ element. The lower picture depicts the swapping of elements within an array. We also have pictorial representations for updating an element with an arbitrary value as well as updating with a value from another element within the array, but space precludes us from presenting them here.

In order to illustrate our basic approach we consider a simple teaching example - the *Polish Flag Problem*. The general idea is to partition a mixture of coloured objects into distinct colours. In the case of the Polish Flag Problem, there are two distinct colours, *i.e.* red and white, corresponding to the colours of the Polish National Flag[2]. A solution to the problem, written in SPARK, is given in Figure 2. Note that an array `Flag` is used to represent the mixture of colours. It is assumed that all the elements of `Flag` are either `Red` or `White`. This assumption is expressed by the following precondition:

```
--# pre (for all I in IndexRange => (Flag(I)=Red or Flag(I)=White));
```

where `IndexRange` defines the range of valid indices for the array `Flag`. The required postcondition takes the following form:

```
--# post for some P in Integer range (Flag'First) .. (Flag'Last+1) =>
--#   ((for all Q in Integer range Flag'First..(P-1) => (Flag(Q)=Red)) and
--#    (for all R in Integer range P..Flag'Last => (Flag(R)=White)));
```

This asserts that on termination all the `Red` elements within `Flag` will proceed the `White` elements, where the existential variable `P` is used to indicate the lower bound of the `White` elements. Note that to accommodate the situation where `Flag` contains no `White` elements, the upper bound of `P` is defined to be

[2] This is a simplification of Dijkstra's *Dutch National Flag Problem* which requires three colours.

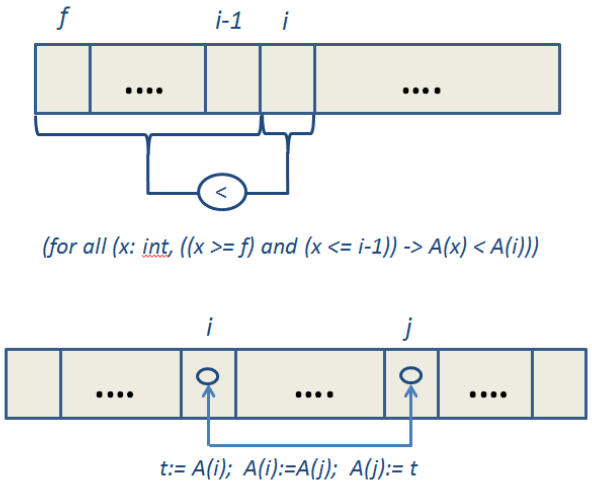


**Fig. 1.** Arrays as Pictures

`Flag'Last+1`. The basic idea behind the algorithm is that a lower segment of `Red` elements and an upper segment of `White` elements are maintained during the computation. Two local variables, `I` and `J` are used in defining the upper and lower bounds of each segment respectively during this computation. Sandwiched between the lower and upper segments (`I..J-1` inclusive) is a mixture of coloured elements. This "basic idea" is expressed formally by the loop invariant corresponding to the assert statement in Figure 2. Note that on termination `I=J` and the consequently mixed colours segment (`I..J-1`) will be empty.

Here we focus on the VCs associated with the loop invariant and the postcondition. With regards to the loop we consider specifically the else-branch, where the corresponding VC is shown in Figure 3. Note that both hypotheses and conclusions are identified using labels prefixed with $H$ and $C$ respectively. Note also that only those parts that are required in order to draw pictures of the array are given. In some sense this is more interesting than the then-branch since the conclusion formulas C4 and C5 involve nested updates, making it harder to decide whether or not the VC is provable. In contrast, we believe that the validity of the VC is more immediate if presented with the pictorial representation as provided in Figure 4. Moreover, we would argue that the picture also provides a strong hint as to how a proof should proceed. That is, it tells you which parts of the goal follow directly from the given, and which parts of the goal must first be decomposed, i.e. the white segment from $j-1$ to $l$ must be decomposed into the $(j-1)^{th}$ element and the segment from $j$ to $l$.

```
...
IndexUpper: constant := 4;
IndexLower: constant := 1;
subtype IndexRange is Integer range IndexLower .. IndexUpper;
subtype PointerRange is Integer range IndexRange'First .. IndexRange'Last+1;
type Colour is (Red, White);
type ArrayOfColours is array (IndexRange) of Colour;
...
procedure Partition_Section(Flag: in out ArrayOfColours)
is
  subtype JustBiggerRange is Integer range Flag'First .. Flag'Last+1;
  I: JustBiggerRange;
  J: JustBiggerRange;
  T: Colour;
  begin
     I:=Flag'First;
     J:=Flag'Last+1;
     loop
        --# assert Flag'First<=I and
        --#        J<=(Flag'Last+1) and
        --#        I<=J and
        --# (for all Q in Integer range Flag'First..(I-1) => (Flag(Q)=Red)) and
        --# (for all R in Integer range J..Flag'Last => (Flag(R)=White));
        exit when I=J;
        if Flag(I)=Red then
           I:=I+1;
        else
           J:=J-1;
           T:=Flag(I);
           Flag(I):=Flag(J);
           Flag(J):=T;
        end if;
     end loop;
end Partition_Section;
```

**Fig. 2.** Solution to Polish Flag problem written in SPARK

Now consider the post-loop VC which is given in Figure 5 and the corresponding pictures shown in Figure 6. Again we argue that the validity of the VC is more immediate when considering the pictorial representation. In addition, the pictures strongly suggest how to complete the proof, i.e. instantiate the existential variable $p$ within the goal to be $i$ (or $j$ since $i = j$).

```
procedure_partition_section_5.
...
H3:    i <= j .
...
H4:    for_all(q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:    for_all(r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
...
H12:   not (i = j) .
...
H17:   not (element(flag, [i]) = red) .
...
        ->
...
C4:    for_all(q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(update(update(flag, [i], element(
           flag, [j - 1])), [j - 1], element(flag, [i])), [
           q_]) = red)) .
C5:    for_all(r_: integer, ((r_ >= j - 1) and (r_ <=
           indexrange__last)) -> (element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [r_]) = white)) .
...
```

**Fig. 3.** Polish Flag: Loop invariant VC - else branch

The real value of pictures, as hinted in the introduction, is in identifying when a VC is not provable or where inconsistencies have arisen between the code and the specification. By way of illustration, consider Figure 7 which gives a revised version of the loop associated with our Polish Flag solution. Here we focus on the verification of the loop invariant with respect to the then-branch. The associated VC is given in Figure 8 while the corresponding pictorial perspective is shown in Figure 9. Again we argue that the pictures are more effective at communicating that there are problems, i.e. the contradiction with regards to the colour of element $i$ within the given hypothesis. This contradiction arises because the loop invariant is flawed, i.e. the upper bound of the red segment should be $(i-1)$ but in the revised loop code it is given as $(i+1)$.

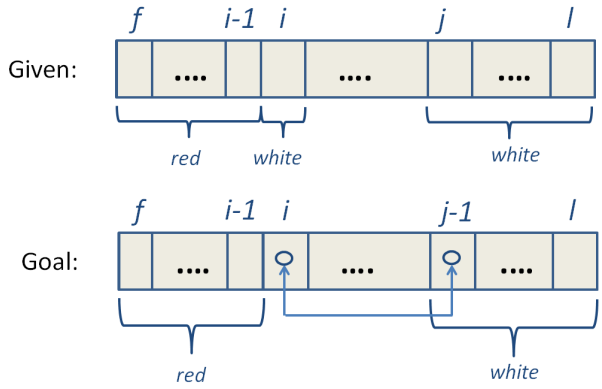


**Fig. 4.** Polish Flag: Loop invariant VC picture - else branch

```
procedure_partition_section_12.
H1:    indexrange__first <= i .
H2:    j <= indexrange__last + 1 .
...
H4:    for_all(q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i - 1)) -> (element(flag, [q_]) = red)) .
H5:    for_all(r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
...
H12:   i = j .
        ->
C1:    for_some(p_: integer, ((p_ >= indexrange__first) and (
           p_ <= indexrange__last + 1)) and ((for_all(q_:
           integer, ((q_ >= indexrange__first) and (q_ <= p_ - 1)) -> (element(
           flag, [q_]) = red))) and (for_all(r_: integer, ((
           r_ >= p_) and (r_ <= indexrange__last)) -> (element(
           flag, [r_]) = white))))) .
```

**Fig. 5.** Polish Flag: Post loop VC

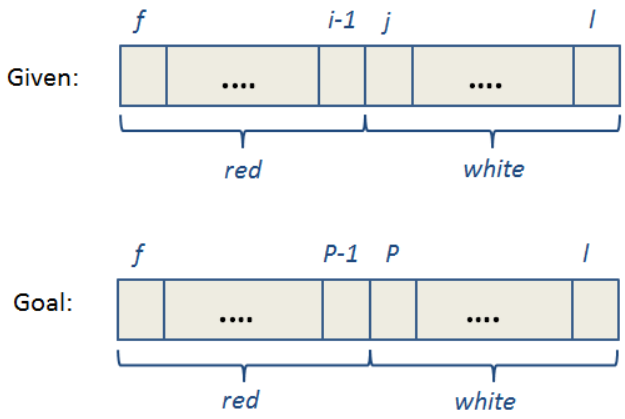


**Fig. 6.** Polish Flag: Post loop VC picture

```
...
loop
--# assert Flag'First<=I and
--#        J<=(Flag'Last+1) and
--#        I<=J and
--# (for all Q in Integer range Flag'First..(I+1) => (Flag(Q)=Red)) and
--# (for all R in Integer range J..Flag'Last => (Flag(R)=White));
         exit when I=J;
         if Flag(I)=White then
            J:=J-1;
            T:=Flag(I);
            Flag(I):=Flag(J);
            Flag(J):=T;
         else
          I:=I+1;
         end if;
end loop;
...
```

**Fig. 7.** Revised Polish Flag code

```
procedure_partition_section_4.
...
H3:    i <= j .
H4:    for_all(q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i + 1)) -> (element(flag, [q_]) = red)) .
H5:    for_all(r_: integer, ((r_ >= j) and (r_ <=
           indexrange__last)) -> (element(flag, [r_]) =
           white)) .
...
H12:   not (i = j) .
...
H17:   element(flag, [i]) = white .
...
        ->
...
C4:    for_all(q_: integer, ((q_ >= indexrange__first) and (
           q_ <= i + 1)) -> (element(update(update(flag, [i], element(
           flag, [j - 1])), [j - 1], element(flag, [i])), [
           q_]) = red)) .
C5:    for_all(r_: integer, ((r_ >= j - 1) and (r_ <=
           indexrange__last)) -> (element(update(update(
           flag, [i], element(flag, [j - 1])), [j - 1], element(
           flag, [i])), [r_]) = white)) .
...
```

**Fig. 8.** Polish Flag: Loop invariant VC - then branch (revised code)

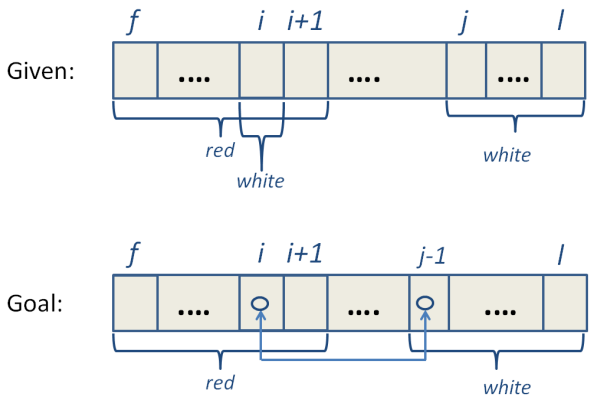


**Fig. 9.** Polish Flag: Loop invariant VC picture - then branch (revised code)

## 4 Experimental implementation and results

We now describe how the basic approach outlined above has been implemented in an experimental tool called Auto-VCV. As shown in Figure 10, Auto-VCV involves three phases:

**Parser:** given a raw VCG file all information relating to arrays is extracted.
**Translator:** from the extracted information the relative ordering of array elements and segments is determined.
**Picture Generator:** the relative ordering information is mapped onto the absolute positioning of the array pictures.

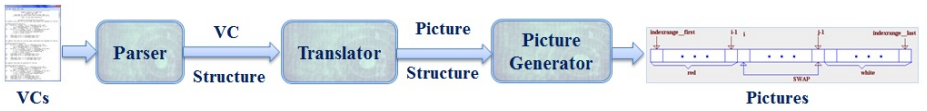


**Fig. 10.** Auto-VCV Architecture

We focus in particular on the core algorithm which extracts information from VCs that is relevant to drawing pictures of arrays. The algorithm takes three input files:

**vcg:** contains all the VCs related to a specific procedure.
**fdl:** records type information as well as the variables and constants associated with the procedure. Any user defined proof functions that are used within assertions are also recorded.
**rul:** contains the definition of proof functions supplied by the user.

Parsing the raw VCs, along with the information in the **fdl**[3] and **rul** files, the algorithm performs the following four tasks for each VC:

1. Identification of the arrays that are explicitly referenced within the given hypotheses and conclusions.
2. Extraction of properties and relations with respect to elements and segments that are contained within the identified arrays, including constraints on index variables and upper and lower bounds.
3. Ordering the elements and segments that are explicitly identified above, this may involve elementary reasoning with regards to the constraints extracted for index variables.
4. Positioning the elements and segments, i.e. determining if segments (and elements) are i) adjoining, ii) non-adjoining, iii) overlapping. Implicit gaps and overlaps are calculated, i.e. either a fixed number of consecutive elements of a segment.

[3] FDL stands for Functional Description Language [1].

The basic tasks outlined above can be applied in two distinct modes within Auto-VCV. Firstly, in what is called *debug mode* pictures are extracted from individual hypotheses (or conclusions) one at a time for each VC. Secondly, in *integrated mode* all the individual pictures are combined to give a single picture for the given VC. The actual picture drawing aspect of the system maps the abstract information extracted from the VCs onto concrete positions within the Auto-VCV interface panels.

Auto-VCV has an object oriented design and is implemented in Java SDK 1.7 version using AWT and Swing utilities along with the Java 2D graphics library [7]. The GUI for Auto-VCV is shown in Figure 11, note that as well as displaying pictures of arrays it also allows the user to view the related VC (bottom panel) and FDL file (bottom right panel). Mode selection and other navigation options are shown in the panel on the right.

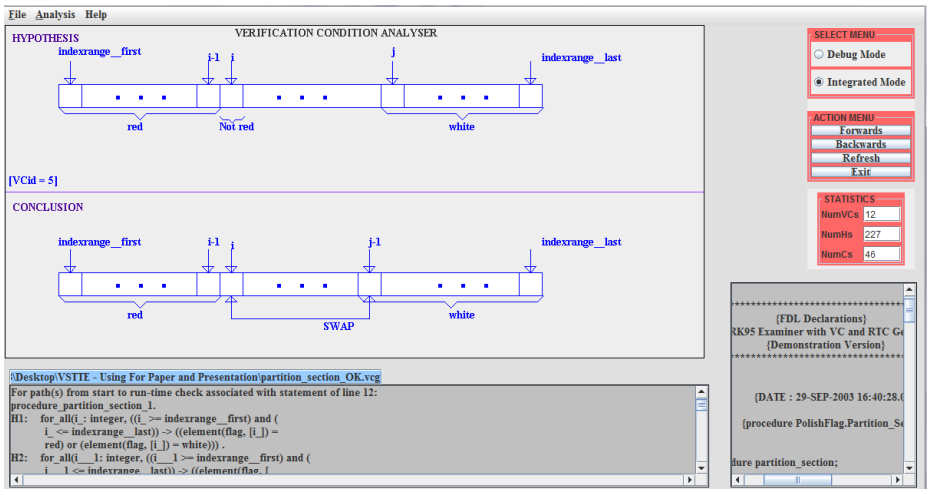


**Fig. 11.** Auto-VCV Screenshot

Returning to our running example, the pictures generated by Auto-VCV for the VC given in Figure 3 are shown in Figure 12, while the pictures generated for the VC given in Figure 8 are shown in Figure 13. In order to illustrate pictures involving relations, consider the `Bubble_Max` procedure given in Figure 14 - a procedure in which the largest value within an array "bubbles" up to the top, i.e. the element with the largest index. The VC associated with the then-branch is given in Figure 15 while the corresponding Auto-VCV generated picture is shown in Figure 16. The VC and pictures associated with the path that avoids the then-branch are given in Figure 17 and Figure 18 respectively. Again we would argue that the validity of these VCs is more immediate when viewed as pictures.

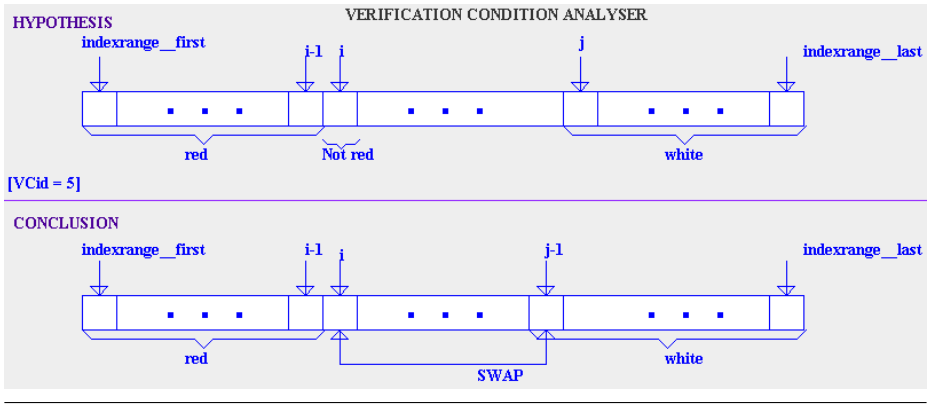


**Fig. 12.** Auto-VCV: Polish Flag loop invariant VC picture - else branch

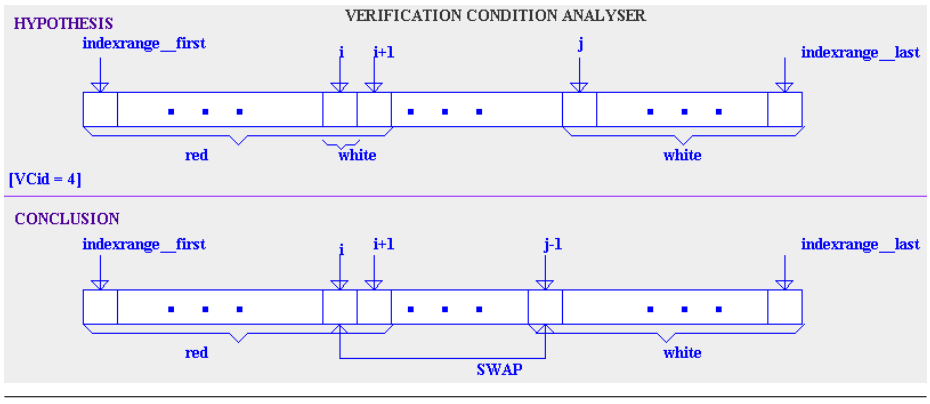


**Fig. 13.** Auto-VCV: Polish Flag loop invariant VC picture - then branch (revised code)

```
subtype Index_Type is Integer range 1 .. 9;
   type Array_Type is array (Index_Type)
   of Integer;
...
procedure Bubble_Max(Table: in out Array_Type)
   is
      R: Index_Type;
      T: Integer;
   begin
      R:= 1;
      loop
 --# assert (for all I in Integer range Table'First .. (R-1) => (Table(I) <= Table(R)));
 exit when R = Index_Type'Last;
    R:=R+1;
            if Table(R-1) > Table(R) then
               T:= Table(R);
               Table(R):= Table(R-1);
               Table(R-1):= T;
            end if;
         end loop;
   end Bubble_Max;
```

**Fig. 14.** Bubble Max code

```
procedure_bubble_max_3.
H1:    for_all(i_: integer, ((i_ >= index_type__first) and (
           i_ <= r - 1)) -> (element(table, [i_]) <= element(
           table, [r]))) .
...
H18:   element(table, [r + 1 - 1]) > element(table, [r + 1]) .
...
        ->
C1:    for_all(i_: integer, ((i_ >= index_type__first) and (
           i_ <= r + 1 - 1)) -> (element(update(update(
           table, [r + 1], element(table, [r + 1 - 1])), [r + 1 - 1], element(
           table, [r + 1])), [i_]) <= element(update(update(
           table, [r + 1], element(table, [r + 1 - 1])), [r + 1 - 1], element(
           table, [r + 1])), [r + 1]))) .
...
```

**Fig. 15.** Bubble Max: Loop invariant VC - then-branch (true)

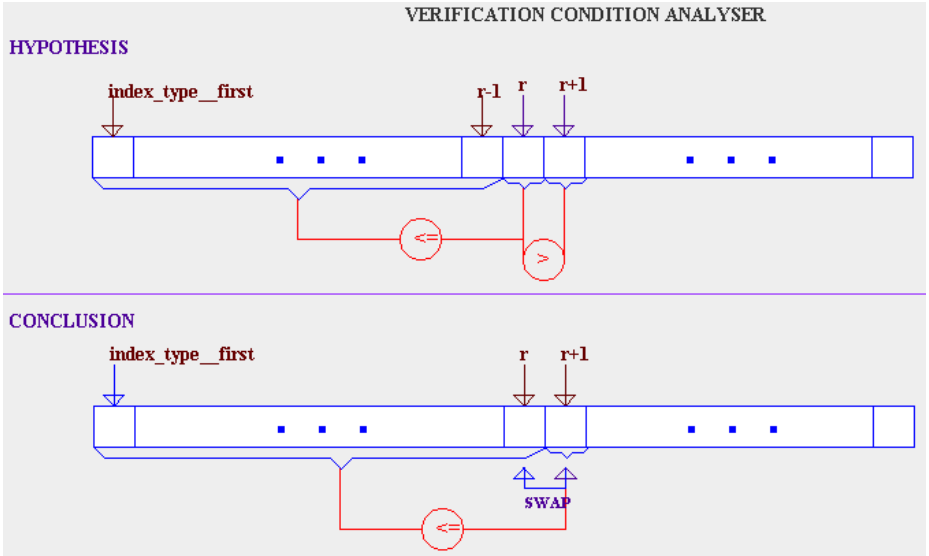


**Fig. 16.** Auto-VCV: Bubble Max loop invariant VC picture - then-branch (true)

```
procedure_bubble_max_4.
H1:    for_all(i_: integer, ((i_ >= index_type__first) and (
           i_ <= r - 1)) -> (element(table, [i_]) <= element(
           table, [r]))) .
...
H18:   not (element(table, [r + 1 - 1]) > element(table, [r + 1])) .
        ->
C1:    for_all(i_: integer, ((i_ >= index_type__first) and (
           i_ <= r + 1 - 1)) -> (element(table, [i_]) <= element(
           table, [r + 1]))) .
...
```

**Fig. 17.** Bubble Max: Loop Invariant VC - then-branch (false)

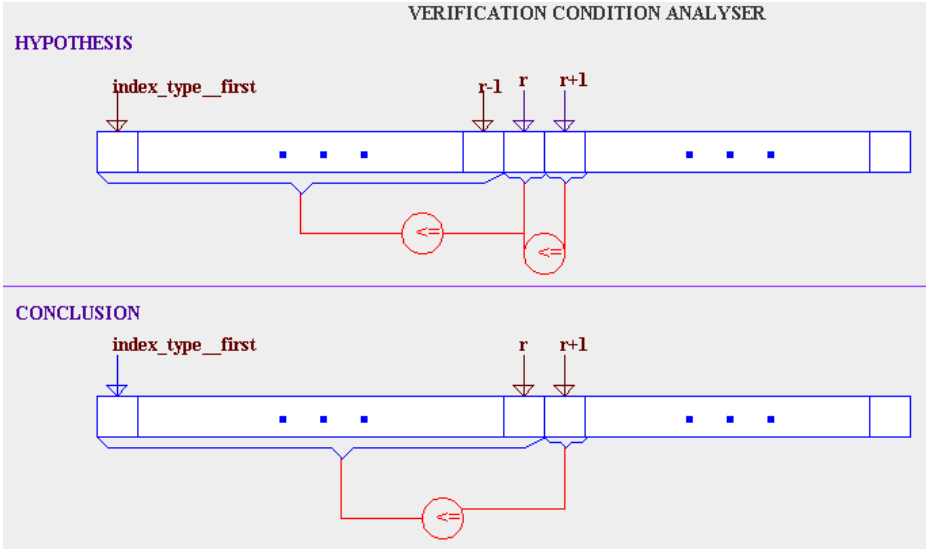


**Fig. 18.** Auto-VCV: Bubble Max loop invariant VC picture - then-branch (false)

## 5 Related and future work

We are unaware of any other work that directly addresses the visualization of array based VCs. As part of a previous project, which focused on separation logic [11], we built an animation tool [10] which supports the visualization of programs that manipulate the heap. The spatial operators associated with separation logic makes it particularly amenable to extracting pictures from formulas.

Further testing and development of the Auto-VCV tool is required. For instance we need to develop the tool so that it can represent relations between distinct pictures, e.g. when proving sorting algorithms one needs to specify that the output array is a permutation of the input array. Moreover, to deal effectively with more comprehensive functional specifications definitions become important. Handling definitions is currently under development, and accounts for the **rul** (file) input to our algorithm discussed above. Multi-dimensional arrays as well as records are also part of our future work plans. Following the motivations of Reynolds [12] mentioned in the introduction, we are also keen to explore the role of pictures within proof.

In terms of SPARK users, we have received positive feedback on Auto-VCV from software engineers within BAE Systems that use SPARK. We also intend to make use of our work within a MSc programme which covers SPARK and program proof. Another potential direction will be to target Boogie, a generic verification condition generator [5]. Following the Boogie route would allow our approach to be more easily applied to other programming languages.

## 6 Conclusion

We have presented an approach to visualizing VCs associated with array based code. The core of the approach has been demonstrated via our Auto-VCV prototype tool which extracts pictures from SPARK VCs. While still very much an experimental tool, we believe that it demonstrates the value of visualizing VCs as pictures, both as an aid to proof as well as debugging code and specifications.

**Acknowledgements** This research was supported by EPSRC Platform Grant EP/J001058. Our thanks go to Alan Bundy, Gudmund Grov, Paul Jackson, Jacques Fleuriot, Ewen Maclean for their feedback on the work, as well as to John Moore and Ben Gorry (BAE Systems, Warton UK) for their feedback on an early prototype of Auto-VCV. We also thank three anonymous VSTTE 2014 referees for their constructive feedback.

## References

1. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.
2. J-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1), 1985.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 3 edition, 2009.
4. M. J. Gordon. *Programming Language Theory and its Implementation.* International Series in Computer Science. Prentice-Hall, 1988.
5. C. Le Goues, K. Rustan M. Leino, and M. Moskal. The boogie verification debugger (tool paper). In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.
6. D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.
7. V. J. Hardy. *Java 2D API graphics.* Sun Microsystems Press Java series. Sun Microsystems Press, 2000.
8. C. A. R. Hoare. Proof of a program: Find. *CACM*, 14(1):39–45, January 1971.
9. C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv*, 41(4), 2009.
10. E. Maclean, A. Ireland, and G. Grov. The core system: Animation and functional correctness of pointer programs. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2011): Tool Demonstration Paper*, Lawrence, Kansas., 2011. IEEE.
11. P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.
12. J. C. Reynolds. *The Craft of Programming.* Prentice-Hall, 1981.
13. N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.