Design, Implementation and Evaluation of MPVS: A Tool to Support the Teaching of a Programming Method

Isabelle Dony

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

September 14, 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Baudouin **Le Charlier** (Advisor)	UCL/INGI, Belgium
Philippe **Delsarte**	UCL/INGI, Belgium
Pierre-Arnoul **de Marneffe**	ULg, Belgium
Kung-Kiu **Lau**	University of Manchester, UK
Jean-François **Raskin**	ULB, Belgium
Yves **Deville** (Chair)	UCL/INGI, Belgium

Contents

Acknowledgements

The realization of this thesis would not have come to an end without the support of many people.

My first thought goes to my advisor, Baudouin Le Charlier, who is the main person responsible for the achievement of this thesis. He has dedicated some considerable time to help me in the redaction of papers and of my thesis; he has provided a lot of helpful comments, advices and suggestions. I thank him for his human qualities and his faculty to regulate my stress. He has transmitted some rigour in my work and some very good teaching influence.

I am also grateful to the members of my jury, Jean-François Raskin, Philippe Delsarte, Pierre-Arnoul de Marneffe, and Kung Kiu Lau, as well as Yves Deville, the president of my jury, for having read my thesis and for their valuable feedback. Thank you Philippe Delsarte for the second reading of my thesis mainly focused on the English language. Special thanks to Yves Deville, for the constant encouragements, especially during the hard times.

Let me also thank Charles Pecheur, for all the discussions we shared, ideas and advices.

I cannot forget to show some gratitude to Raphaël Collet. His advices are the bases of my work. Without him, I certainly would never have used the Oz language in my thesis. He has spent quite some time to help me and all his free consulting days are priceless. Many thanks for his availability.

I also extend my gratitude and appreciation to many other colleagues, previous and current. Donatien shared my office. He helped me in several ways, from Oz consultant to friend relation. Marie-Catherine also worked several years right in front of me. We shared many enthusiastic discussions about teaching, but also about the life in general. Thank you Nicolas and Xavier, Gustavo and François, my next door office neighbours, to have helped me so often, solving technical problems, or sharing my recurrent doubts about this thesis... Thank you to my newly colleague José, who has used the preliminary version of my tool for his memory. He was very helpful for several improvements, but still more thank for the numerous discussions about teaching: for the past, but also the coming year, a year full of questions... I wish him good success with his thesis. Special thanks to the feminine minority of the department, who was there to cheer me up...

Thank you to all the students and especially the three grade students who have experimented our tool.

I would like to thank our close group of friends (le souper piscine) for their great sense of humour, their support during all these years, and the great moments we have spent together.

Many thanks to my parents and parents in law, for their constant presence each time I needed some help. They allow me to manage both my thesis and my familial life. Special thanks to Francis, for his careful readings.

Of course, I cannot close this acknowledgement section without thanking with all my heart my husband Valery Anciaux, for his love and tireless support during my whole thesis. Living by my side during my doubts and my existential questions was not a piece of cake. Thanks a lot for his several reading and re-reading of my thesis.

My thanks also go to my children Corentin and Charlotte who regularly brought me down-to-earth...

I promise you all, I will never bother you again with my thesis.

Abstract

Teaching formal methods is notoriously difficult and is linked to motivation problems among the students; we think that formal methods need to be supported by adequate tools to get better acceptance from the students. One of the goals of the thesis is to build a practical tool to help students to deeply understand the classical programming methodology based on specifications, loop invariants, and decomposition into subproblems advocated by Dijkstra, Gries, and Hoare to name only a few famous computer scientists. Our motivation to build this tool is twofold. On the one hand, we demonstrate that existing verification tools (e.g., ESC/Java, Spark, SMV) are too complex to be used in a pedagogical context; moreover they often lack completeness, (and sometimes, even soundness). On the other hand teaching formal (i.e., rigorous) program construction with pen and paper does not motivate students at all. Thus, since students love to use tools, providing them with a tool that checks not only their programs but also their specifications and the structure of their reasoning seemed appealing to us.

Obviously, building such a system is far from an easy task. It may even be thought completely unfeasible to experts in the field. Our approach is to restrict our ambition to a very simple programming language with simple types (limited to finite domains) and arrays. In this context, it is possible to specify problems and subproblems, both clearly and formally, using a specific assertion language based on mathematical logic. It appears that constraint programming over finite domains is especially convenient to check the kind of verification conditions that are needed to express the correctness of imperative programs. However, to conveniently generate the constraint problems equivalent to a given verification condition, we wish to have at hand a powerful language that allows us to interleave constraints generation, constraints solving, and to specify a distribution strategy to overcome the incompleteness of the usual consistency techniques used by finite domain constraint programming. We show in this thesis that the Oz language includes all programming mechanisms needed to reach our goals.

Such a tool has been fully implemented and is intended to provide interesting feedback to students learning the programming method: it detects programming and/or reasoning errors and it provides typical counter-examples. We argue that our system is adapted to our pedagogical context and we report on experiments of using the tool with students in a third year programming course.

Part I

A Pedagogical Challenge

Chapter 1

Introduction

One of the big challenges when teaching formal methods is to motivate students. This difficulty is by no means new since, in 1981, Dijkstra wrote in his foreword to Gries book [26]: *we get a glimpse of educational challenge we are facing: besides teaching technicalities, we have to overcome the mental resistance always evoked when it is shown how the techniques of scientific thought can be fruitfully applied to a next endeavour.* A quarter of a century later, the discipline of computer science is more mature, the discipline of Formal Methods has evolved, but the educational challenge appears to be as great as ever. Since Dijkstra, the boom of information technology has left aside the rigorous programming methods. Formal methods have mainly focused on practical aspects, like embedded systems and static detection of programming errors. From our side, we think it is essentiel for students to approach a good and rigorous programming method, even if it is not directly based on a practical field. Even if students are more motivated when they write a concrete application, it is by learning to write correct programs that they will master the art of working out concrete applications. Our two main objectives are to motivate our students, and to find a better approach to make them understand what a good programming method is.

In such a context, the idea is to make an experimentation with students. With our teaching experience, the idea is to enhance our course with a tool to motivate the students to learn the structured programming method. We would like to help the students to overcome *the mental resistance*: using an appropriate software could enforce the use of the method, give a precise feedback to the students, help them to understand their reasoning errors. The requirements for such a tool have already been studied in [14]. However, building such a system is far from an easy task. It may even be thought completely unfeasible to experts in the field, since verifying the correctness of algorithms is an undecidable problem in the general case. We think that such a tool can be implemented if we restrict our ambition to our pedagogical context and if we consider a simple programming language.

1.1 Our Contributions

Analysis of how existing tools can help to teach structured programming There are two complementary approaches to program verifications: model checking and theorem proving. Model checking is a formal verification technique based on state exploration. Making an exhaustive verification can give a precise result claiming that the property is true or else providing counter-examples. However, the size of the state space is often exponential in the size of the system description, resulting in the state explosion problem. Theorem proving uses a mechanical verification, it does not have the constraints on the domain size but it is not complete and/or it requires user's guidance. Among tools based on theorem proving, we have considered ESC/Java and SPARK. ESC/Java [23, 12, 16] has been designed to be fully automatic. As a consequence, it focuses on some limited properties, and is nor sound nor complete. SPARK[3] is a powerful tool, used in mission critical applications. But it requires extensive human guidance. We also studied a model checker: SMV [44]. It gives an interesting feedback about the correctness of algorithms, but it cannot be used directly by the students, because encoding the programs and the verification conditions into SMV is not a trivial work. In conclusion, none of these tools are exactly appropriate to fulfill our needs because they require too much expertise and often do not provide precise feedback about the verification.

Definition of a new tool devoted to teach structured programming In our programming learning context, we would like a fully automatic tool, easy to manipulate, enforcing the use of the structured programming method and giving precise feedback to help the students to understand their reasoning errors. The ideal tool should be easy to use, while applying the non trivial concepts taught. This should enforce rigor, the role of the invariant and the decomposition into subproblems. The feedback given by the tool should be clear, to help our students to understand their reasoning errors. Displaying precise counter-examples may be very informative, and false warnings and forgotten errors have to be avoided. Our approach is to restrict our ambition to a simple programming language with simple types (limited to finite domains) and arrays. We do not need to cover a large and complex programming language like Java: we consider that, for teaching algorithmic, we do not need a very elaborate language. The simpler it is, the better it is. In this context, it is possible to specify problems and subproblems, both clearly and formally, using a specific assertion language based on mathematical logic: the specification language can be expressive enough to express as directly as possible what the student means.

Development of constraint programming techniques to implement the new tool In this restricted context, it appears that constraint programming over finite domains is especially convenient to check the kind of verification conditions that are needed to express the correctness of imperative programs. To conveniently generate the constraint problems equivalent to a given verification condition, we use the Oz language, a multiparadigm and powerful language.

Experiments with the tool in an advanced programming course Such a tool has been fully implemented. In the context where it is intended to provide interesting feedback to students learning the programming method, we analyse the impact of such a tool on the behaviour, the motivation and the understanding of the students learning this method. We have the opportunity to participate in the "Program Conception Methods" course [39] addressed to the computer science students in the third year of their studies. This tool has been experienced twice as a support of this course.

1.2 Overview of the Thesis

- **Chapter 2: Teaching a Structured Programming Method: Principles and Difficulties**

  We describe the main aspects of structured programming based on Baudouin Le Charlier's presentation [37]. We present the reasoning methods, considering the constructive approach and the verification approach. To illustrate these, we have chosen four examples, with increasing difficulties; for each of them, we detail the process of constructing the algorithm and, thereafter, we make the mathematical proofs of correctness. These examples are used later on for discussing program verification with tools. We conclude with the pedagogical findings of this method. We believe that supporting this method with an adapted software can help the students: we enumerate the requirements of such a tool.

- **Chapter 3: How about Existing Tools to Teach Structured Programming?**

  We describe three existing verification tools that could be interesting to use in our pedagogical context: ESC/Java2, SPARK, and SMV. For each system, we first describe the technical aspect, we present some scenarii of verification with the algorithms we have studied in Chapter 2, and we conclude that it is difficult to find an appropriate existing tool in our pedagogical context. Finally, we argue that the best way to have a precise feedback about the correctness of the programs, is to perform exhaustive verification even if we have to restrict the program variable domains to small finite domains.

- **Chapter 4: A new Tool to Teach Structured Programming**

  We present the tool that we have completely implemented. We show several scenarii of verification with the same algorithms as in the preceding chapters. We discuss the advantages comparing to the studied existing tools.

- **Chapter 5: Definition of Languages** To deeply understand the behaviour of the tool and to have a reference for the next chapter which explains the implementation, we give a complete definition of the programming language and the assertion language accepted by our tool.

- **Chapter 6: Implementation of the Tool using the Mozart Programming System**

  We describe the way verification conditions are checked using finite domain constraint programming. We show how our programs are translated into contraints and how counter-examples are provided.

- **Chapter 7: Using our Tool in a Programming Course**

  We present two experimentations with the tool, and we conclude with our feeling about the impact of this tool on the student motivation and about their understanding level.

- **Chapter 8: Conclusion**

  We finally conclude about this thesis project. Working out such a tool is possible and can help to teach a structured programming method. However, the tool has some drawbacks that are inherent in any tool.

Chapter 2

Teaching a Structured Programming Method: Principles and Difficulties

2.1 A Method of Structured Programming

[1] A while ago, the programming task did not involve much theoretical concern. Learning how to program was restricted to learning a programming language and few people worked on program correctness. The first article on proving program correctness [46] was by Peter Naur in 1966. Naur emphasised the importance of program proofs and provided an informal technique for their specification. Then, work on program correctness has moved up: Robert Floyd [24] attached assertions to programs, Tony Hoare [28] defined his well-known theory: it proves partial correctness of programs by defining a restricting programming language in terms of a logical system of axiom and inference rules. From this, a lot of research on axiomatisation has been done [30, 31, 32]. In 1976, Edger W.Dijkstra [17] introduced weakest preconditions and then showed, through examples, how they could be used as a "calculus for the derivation of programs". From this time, it became clear that theory and formalism could actually lead to development of programs in a more reliable manner. A programming discipline with a method based on formal logic was born. Its goal was to learn writing a program from the specifications:
Specifying, then *constructing* the program by *proving* that it is correct. This approach is based on rigour, reasoning, loop invariant, decomposition into subproblems, and formal specification.

[1] based on the book of D. Gries [26]

In this chapter, we describe the main aspects of structured programming based on Baudouin Le Charlier's presentation [37]. First, we introduce basic notions needed for writing simple algorithms. Next, we present the reasoning methods. We consider the constructive approach and the verification approach. To illustrate them, we have chosen four examples, with increasing difficulties; for each of them, we detail the process of constructing the algorithm and then we make the mathematical proofs to ensure the correctness of the algorithm. These examples will be used later on for discussing program verifications with tools.

2.1.1 Basic Notions Related to Algorithms

An algorithm is an ordered set of precise rules, used to produce some output when the input data is correctly chosen. J.Arsac [2]

This definition needs to be contextualized: in our programming learning context, we first introduce the objects we are going to manipulate (the *data* and the auxiliary objects), then we detail the basic operations and the control structures to be used to define and organise the *rules*. Next, we explain what a correct algorithm is, and the meaning of *input data correctly chosen* and *output*.

We use integer and Boolean values, simple programming variables (of integer or Boolean type) and integer constants; we also manipulate arrays: $a[u..v]$ where u and v are constants. It means that the indexed variables $a[u]$, $a[u+1]$,..., $a[v]$ exist; but evaluating an indexed variable $a[i]$ with $i > v$ or $i < u$ returns an error. $a[u..v]$ with $v < u$ is an empty array.

The basic components are *expressions*, *conditions* and *assignments*. The simplest expressions are values, simple variables or indexed variables. The more complex expressions involve operations. For arithmetic expressions, we allow addition, subtraction, multiplication, integer division and the modulo operation. For Boolean expressions (conditions), the allowed operations are comparisons between integer expressions, and equality or inequality between Booleans expressions. Global operations on arrays are not allowed. When evaluating an expression, the values of the involved variables are read but never modified. Naturally, to have a well-defined evaluation, all the variables involved must be initialised. In a first approach, we consider that the integer domain is infinite, and so we avoid, at this moment, some out of domain bounds problems. The execution of an assignment, `x:= expr` modifies or initialises the variable `x` with the value of the right expression `expr`.

We define a *statement* as an action, or a set of actions on the program variables. The simplest one is the assignment. For the other statements, we use three control structures: *sequence of statements*, *conditional statement* and *while statement*.

```
   S1;                 if C then S1          Init;
   S2;                      else S2          while not H
   ..                  end                      Iter
   SN                                        end;
                                             Clot
```

sequence of statements   conditional statement   while statement

Here `S1`, `S2`, ...,`SN` stand for simpler statements, they can be themselves sequences, conditional statements or while statements. `C` is a condition (Boolean expression) and according to the evaluation of `C`, the statement `S1` or the statement `S2` is executed. In a while statement, we assign specific roles to the statements: `Init` stands for the *initialisation*, `Iter` stands for the *iteration* and `Clot`, for the *closure*. `H` is the *halting condition*.

After having presented all the required elements of an algorithm, let us define what a correct algorithm is. An algorithm is *correct according to its specification*: an initial situation (the precondition) and a final situation (the postcondition). When the input data satisfies the precondition, the algorithm must guarantee that the output fulfills the postcondition. In fact, the final situation is reached through successive modifications from the initial situation. Initial, final and intermediate situations are named *assertions* when they are expressed in a mathematical way.

Here is an example of a sequence of assignments. We can observe the different concepts that we have introduced. A standard format is set for the used variables and the pre and postconditions. Besides, we write assertions between braces since they need to be distinguished from the code.
The goal of the algorithm is to exchange the values of two variables, a and b.

- Declarations:

  var a, b : integer;

- Pre: a, b initialised

  Post: $a = b_0$ and $b = a_0$

- $\{a = a_0$ and $b = b_0\}$

  `t := a;`

```
{$a = a_0$ and $b = b_0$ and $t = a_0$}
a := b;
{$a = b_0$ and $b = b_0$ and $t = a_0$}
b := t
{$a = b_0$ and $b = a_0$}
```

We can notice some successive modifications of the initial assertion before reaching the postcondition. By convention, a_0 and b_0 represent the initial values of a and b.

For an iterative algorithm, we need an *invariant*, a general situation encapsulating all intermediate situations we get at each iteration. In other words, the invariant is an assertion that must hold before and after every iteration.

- Declarations:

  const n; $n \geq 0$;

  tab a: array[1..n] of integer ;

  var s: integer;

- Pre: a initialised

  Post: a is unchanged and $s = \Sigma_{j=1}^{j=n} a[j]$

-
```
{$a$ is initialised}
i := 1 ;
s := 0 ;
{$a = a_0$ and $1 \leq i \leq n + 1$ and $s = \Sigma_{j=1}^{j=i-1} a[j]$}
while (i <> n + 1)
      s := s + a[i] ;
      i := i + 1 ;
end
{$a = a_0$ and $s = \Sigma_{j=1}^{j=n} a[j]$}
```

This example computes the sum of the array elements; it shows us that assertions can be more expressive than the algorithm expressions and conditions. As we will see in the following chapters, we can write less or more formal assertions depending on the context of reasoning about an algorithm.

In our reasoning about algorithms, we use the Hoare notation: let Q, and P be two assertions, and let S be a statement, we write

$$\{P\}\ S\ \{Q\}$$

to mean that if the assertion P holds before executing S, it is guaranteed that S terminates and does not generate any run-time error, and that after executing S, the assertion Q also holds. Notice that we use the notation to express *total correctness*, not only partial correctness as in Hoare's original proposal [29]; partial correctness simply requires that *if* S terminates and if no runtime error occurs, then it is correct. In our view, S must terminate. Moreover, it must terminate without runtime error.

In the specific context of the while statement, we have:

- $\{Pre\}\ Init\ \{Inv\}$
- $\{Inv$ and not $H\}\ Iter\ \{Inv\}$
- $\{Inv$ and $H\}\ Clot\ \{Post\}$

To have total correctness of a while statement, we need the total correctness of the three Hoare propositions, and besides, we need to guarantee the termination of the loop by proving a *variant*, a positive integer function of the program variables that strictly decreases at each iteration.

In the next sections, we develop two reasoning approaches based on these notations.

- In **the constructive approach**, we first define the pre and post conditions; if there is no obvious way to go from the precondition to the postcondition, we search for intermediate assertions that seem to make the problem easier to solve, then we derive the statements. If the process is iterative, we choose an invariant and then we derive the initialisation, the iteration, the closure and halting condition.

  Elaborating the problem by reasoning with intermediate situations leads us to reduce the problem to very simple problems. The benefit of this method stems from the divide-and-conquer approach. This method of construction of algorithm guarantees the correctness of the algorithm since the subproblems are very simple to solve.

- In the **verification approach**, giving the code and the assertions (invariant and other intermediate assertions), we mathematically prove the correctness of each transformation $\{P\}S\{Q\}$.

In each of the following examples, we first present in a concrete way the construction method by pinpointing some difficulties that we (more specifically students) generally meet, then we present some interesting proofs. The typical patterns are represented through these examples.

2.1.2 The Indian Exponentiation Algorithm

The first example uses simple variables, and the invariant expresses an interesting relation. The first part of the exercice consists in elaborating an algorithm computing x^y in $\log(y)$ time where $y \geq 0$, assuming that $0^0 = 1$. Then, we prove the correctness of the algorithm according to the specifications.

Constructing a correct algorithm using the invariant method First, in order to explain the intuitive reasoning using concrete values, let us compute 10^{15}:

$$\begin{array}{rcll}
10^{15} & = & 10 & * \; 10^{14} \\
 & = & 10 & * \; (10^2)^7 \\
 & = & 10 & * \; 100^7 \\
 & = & 10 * 100 & * \; 100^6 \\
 & = & 1000 & * \; (100^2)^3 \\
 & = & 1000 & * \; 10000^3 \\
 & = & 1000 * 10000 & * \; 10000^2 \\
 & = & 10000000 & * \; 100000000 \\
 & = & 1000000000000000 &
\end{array}$$

From this, we hightligth a constant relation $x^y = z * u^v$ where z, u and v are auxiliary variables that lead us to the result that will be in z (when $v = 0$). Now, we are able to fix the declarations; we can specify the precondition and the postcondition, and identify the invariant from the relation that we have highlighted.

These first steps are depicted below.

- Declarations:

  var x, y : integer; (input)

  var z : integer; (output)

  var u, v : integer; (auxiliary variables)

- Pre: x, y initialised and $y \geq 0$

  Post: x, y unchanged and $z = x^y$

  Inv: x, y unchanged and $v \geq 0$ and $x^y = z * u^v$

When these steps are completed, we derive the statements from the Hoare propositions.

First, we derive the initialisation from $\{Pre\}$ $Init$ $\{Inv\}$.
We want to get $x^y = z * u^v$.
It is clear that if we choose $z = 1$ and $u = x$ and $v = y$, this equality is satisfied.
To have this, we simply need to execute the three statements

```
z := 1 ; u := x ; v := y
```

which are correct because the precondition guarantees that x and y are initialised.
Besides, $v \geq 0$ because $y \geq 0$.

Then, we need to discover the halting condition and we derive the iteration from $\{Inv$ and not $H\}$ $Iter$ $\{Inv\}$.

The condition `v=0` seems a judicious halting condition because if $v = 0$, then $x^y = z * u^v$ becomes $x^y = z$, i.e., the relation of the postcondition. So, before each iteration, we have $x^y = z * u^v$ with $v > 0$ and we want $x^y = z * u^v$ by modifying the auxiliary variables z, u, v. Let $u = u_1$, $v = v_1$, $z = z_1$, satisfying $x^y = z_1 * u_1^{v_1}$ and $v_1 > 0$.

1. If v_1 is even, then

   $z_1 * u_1^{v_1} = z_1 * (u_1^2)^{v_1 \text{ div } 2}$

   So we restore the invariant such that $z = z_1$ and $u = u_1^2$ and $v = v_1$ div 2 by executing

   ```
   u := u * u ; v := v div 2
   ```

   which are correct statements because v and u are initialised and the division operation does not divide by zero.

   We get the equality $x^y = z * u^v$, and v stays positive because $v_1 \geq 2$.

2. If v is odd, then,

   $z_1 * u_1^{v_1} = z_1 * u_1 * (u_1^2)^{(v_1-1) \text{ div } 2}$

   with $z = z_1 * u_1$ and $u = u_1^2$ and $v = (v_1 - 1)$ div 2, we satisfy the invariant.

   We can execute

   ```
   z := z * u ; v := v - 1 ; u := u * u ; v := v div 2
   ```

   which are correct for similar reasons as in the previous item (when v is even).

   We observe that v stays positive because `v := v - 1`, $v_1 > 0$.

Next, no closure is needed to have $\{Inv$ and $H\}$ $Clot$ $\{Post\}$, because the postcondition is exactly the result obtained when leaving the loop: ($x^y = z * u^v$ and $v = 0$) is equivalent to $x^y = z$.

Finally, we represent in Figure 2.1 the algorithm in the loop format.

- Declarations:

  var x, y : integer; (input)

  var z : integer; (output)

  var u, v : integer; (auxiliary variables)

- <u>Pre:</u> x, y initialised and $y \geq 0$

  <u>Post:</u> x, y unchanged and $z = x^y$

-
```
u := x ; v := y ; z := 1
while (v <> 0)
    if (v mod 2 = 1) then
         z := z * u ;
         v := v - 1
    end
    u := u * u ;
    v := v div 2
end
```

Figure 2.1: The algorithm of the Indian exponentiation

Proving the correctness of the algorithm To prove $\{P\}\ S\ \{Q\}$, we make a symbolic execution. It means that we give initial symbolic values to the variables used by S. Then, for all possible execution paths of S, we calculate the final values, we verify that the execution is well defined and that the final values satisfy Q.

We have to prove three Hoare propositions:

- $\{Pre\}\ Init\ \{Inv\}$

  After the initialisation execution, $x^y = z * u^v$ is correct.

  Indeed, the equality is trivial with $u = x$, $v = y$ and $z = 1$.

- $\{Inv$ and not $H\}\ Iter\ \{Inv\}$

  Let u_1, v_1, z_1 be the initial values of u, v, z;

  by hypothesis, $z_1 * u_1^{v_1} = x^y$ and $v_1 > 0$.

  To verify that the invariant holds after the iteration, we have to distinguish two cases generated by the conditional statement.

  Before, we verify that the condition `(v mod 2 = 1)` is well-defined: v_0 is initialized and we do not have a modulo operation by zero.

  1. If v_1 mod $2 = 0$ then

     we immediately see that no runtime error can occur because u and v are initialised.

     So, the execution of the iteration terminates with

     $v = v_1$ div 2, $u = u_1 * u_1$, $z = z_1$.

The invariant holds:

$v \geq 0$ because

$$v = v_1 \text{ div } 2 \geq 0 \text{ div } 2 = 0$$

and

$$\begin{aligned} z * u^v &= z_1 * (u_1 * u_1)^{v_1 \text{ div } 2} \\ &= z_1 * (u_1^2)^{v_1 \text{ div } 2} \\ &= z_1 * u_1^{v_1} && (v_1 \text{ is even}) \\ &= x^y && (\text{by hypothesis}) \end{aligned}$$

2. If v_1 mod $2 = 1$ then

   we immediately see that no runtime error can occur for similar reasons as in item 1.

   So, the execution of the iteration terminates with

   $v = (v_1 - 1)$ div 2 , $u = u_1 * u_1$, $z = z_1 * u_1$.

   The invariant holds:

   $v \geq 0$ because

   $$v_1 \geq 1 \text{ and } v = (v_1 - 1) \text{ div } 2 \geq (1 - 1) \text{ div } 2 = 0$$

   and

   $$\begin{aligned} z * u^v &= (z_1 * u_1) * (u_1 * u_1)^{(v_1 - 1) \text{ div } 2} \\ &= (z_1 * u_1) * (u_1^2)^{(v_1 - 1) \text{ div } 2} \\ &= (z_1 * u_1) * u_1^{v_1 - 1} && (v_1 \text{ is odd}) \\ &= z_1 * u_1^{v_1} \\ &= x^y && (\text{by hypothesis}) \end{aligned}$$

- $\{Inv$ and $H\}$ $Clot$ $\{Post\}$

  the closure is empty and,

  by hypothesis, $x^y = z * u^v$ and $v = 0$.

  Hence the postcondition is satisfied since $x^y = z * u^0 = z$.

Finally, proving the three Hoare propositions only guarantees partial correctness of the algorithm. To prove total correctness, we still have to prove the loop termination by providing a *variant*. A variant is a positive integer function of the program variables that strictly decreases. In this example, we can easily see that v is an appropriate variant, since v strictly decreases to 0.

More formal reasoning on the assertions can be made using, for example, the *strongest postcondition* or the *weakest precondition* method. They are less intuitive than the symbolic execution method and they require much more formalisation to be used; for small algorithms using simple variables, they can be used, but for more complex algorithms with complex assertions, these methods are not easy to use. Actually, they are mainly used by automated verification tools; it is in this context that we mention these methods later, in the next chapters. These methods are also in the student cursus but we do not give them priority: our main goal is to teach a method of construction of algorithm with a reasonable formalism and not to manipulate difficult formulas.

2.1.3 The Binary Search Algorithm

Now, we use the arrays in the classical binary search algorithm. This is an algorithm notoriously difficult to construct correctly without a good programming method.

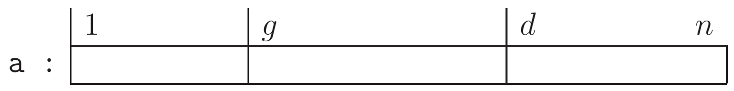
Constructing a correct algorithm using the invariant method The goal of the algorithm is to look for the value of x in the increasing array a by binary search. If the value of x occurs in a, the Boolean result variable b is set to true; otherwise, b is set to false.
We first express the specifications:

Precondition The array a is initialised and is sorted (increasing).

Postcondition The variable b is equal to true if x occurs in the array a; it is equal to false otherwise.

This is how the binary search principle works: it splits a sorted subarray that may contain the element x in two parts. Then, according to the value of the middle element of this subarray, we keep the half part that may contain x. By iterating this process, we can easily see that the array stays always divided in three parts.

It is time to choose a loop invariant: to divide the array a, we use two indices g and d. The key of success to have a correct algorithm is the enforcement of an index convention. We choose g, the index of the first element out of the left part of the array and d, the index of the first value of the right part. The invariant is:



$1 \leq g \leq d \leq n+1$
x is strictly greater than any value of $a[1..g-1]$
x is strictly smaller than any value of $a[d..n]$
a is unchanged

Now, we are able to derive the statements:

According to the invariant picture, the statements are the following

```
g := 1 ; d := n+1
```

Indeed, the left and right examined parts are empty, and `x` cannot have been found in `a`.

For the iteration, we choose an element in the middle of the subarray $a[g..d-1]$, namely $a[m]$ (this subarray cannot be empty i.e., $g < d$). To determine m, we compute $m = (g+d)$ div 2; we verify that $a[m]$ belongs to $a[g..d-1]$ (which also guarantees that $a[m]$ belongs to $a[1..n]$):

1. $m < d$ since

$$\begin{array}{lcll} (g+d) \text{ div } 2 & \leq & (g+d)/2 & \text{(div takes the quotient by default)} \\ & < & (d+d)/2 & = d \end{array}$$

2. $g \leq m$ since

$$(g+d) \text{ div } 2 \geq (g+g) \text{ div } 2 = g$$

In fact, according to the picture, we have four possibilities:

1. $g = d$: the execution can stop, b must be false; the statement is

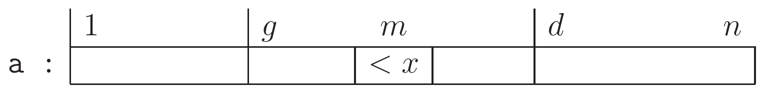
```
b := false
```

2. $a[m] = x$ (and $a[g..d-1]$ is not empty):

   x has been found and the execution can stop, b must be true; the statement is

```
b := true
```

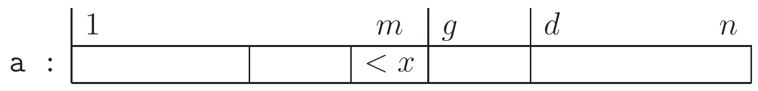
3. $a[m] < x$ (and $a[g..d-1]$ is not empty):



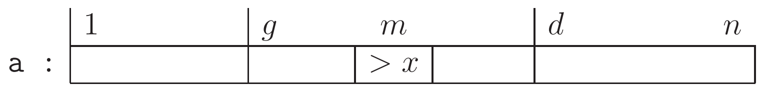
   the statement is

```
g := m + 1
```

   and all the elements of the new left subarray $a[1..g-1]$ are still strictly smaller that x:



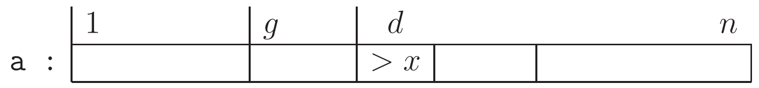
4. $a[m] > x$ (and $a[g..d-1]$ is not empty):



the statement is

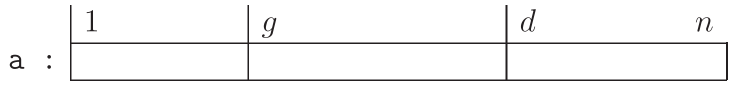
```
d := m
```

and all the elements of the new subarray $a[d..n]$ are still strictly greater than x:



So the iteration terminates when the subarray $a[g..d-1]$ is empty or when $a[m] = x$.

For conception reasons, we choose that b stands in the halting condition to say whether $a[m] = x$ or not. So, the halting condition becomes `b = true or g = d`, and the invariant has to be modified:



$1 \leq g \leq d \leq n+1$
x is strictly greater than any value of $a[1..g-1]$
x is strictly smaller than any value of $a[d..n]$
a is unchanged
b is true if x has been found in $a[1..n]$

The initialisation changes:

```
g := 1 ; d := n+1 ; b:= false
```

The iteration becomes

```
 m := (g + d) div 2 ;
if(a[m] < x) then g := m + 1 ;
if(a[m] > x) then d := m ;
if(a[m] = x) then b := true
```

No closure is needed, because the value of b satisfies the postcondition when the iteration stops with $b = true$ and when the iteration stops with $g = d$.
Together, the invariant and the halting condition say that:
if $b = true$, then x has been found in $a[1..n]$;
if $g = d$, then we know that x does not occur in a because all the elements on the left of g are strictly smaller than x and all the elements on the right of g (including $a[g]$) are strictly greater than x.

A complete version of the algorithm with declarations, more formal precondition and postcondition is given in Figure 2.2.

- Declarations:
  const n;
  var a: array[1..n] of integer; x: integer; (input)
  var b: Boolean; (output)
  var g, d, m: integer; (auxiliary variables)
- Pre: a, x initialised, n is the array size, $(\forall\ i : 1 \leq i \leq n-1 : a[i] \leq a[i+1])$
  Post: a, x unchanged, $b = (\exists\ i : 1 \leq i \leq n : a[i] = x)$
-
```
g := 1 ; d := n + 1 ; b := false
while (not (g = d or b))
      m := (g + d) div 2 ;
     if(a[m] < x) then g := m + 1 ;
     if(a[m] > x) then d := m ;
     if(a[m] = x) then b := true
end
```

Figure 2.2: A binary search algorithm and its specification

Before doing a mathematical proof of the correctness of the algorithm, we need to formalise the invariant picture:

(1) $1 \leq g \leq d \leq n+1$
(2) $(\forall\ i : 1 \leq i < g : a[i] < x)$
(3) $(\forall\ i : d \leq i \leq n : a[i] > x)$
(4) $b \Rightarrow (\exists\ i : 1 \leq i \leq n : a[i] = x)$
(5) a unchanged

Line 4 formally translates the role of b described in the informal invariant: when b is true, we guarantee that $(\exists\ i : 1 \leq i \leq n : a[i] = x)$; when b is false, we cannot guarantee there is no x in a unless when $g = d$: in this case, lines 2 and 3 assert that x does not accur in a, indeed, x is neither in the left subarray of a, nor in the right subarray.

Proving the correctness of the binary search algorithm Focusing on the iteration, let us verify that the proposition $\{Inv$ and not $H\}$ $Iter$ $\{Inv\}$ holds.

Let g_1, d_1, b_1 be the initial values of g, d, b;
by hypothesis,

(1) $(\forall\ i : 1 \leq i < n : a[i] < a[i+1])$
(2) $1 \leq g_1 < d_1 \leq n+1$
(3) $(\forall\ i : 1 \leq i < g_1 : a[i] < x)$
(4) $(\forall\ i : d_1 \leq i \leq n : a[i] > x)$

(5) $b_1 \Rightarrow (\exists\ i : 1 \le i \le n : a[i] = x)$
(6) a is unchanged
(7) $b_1 = false$

Notice that hypothesis **(5)** is true because of **(7)**.
After the first statement, $m = (g_1 + d_1)\ div\ 2$.

1. There are no out of bound error for $a[m]$:

   (a) $m \le n$
   since $(g_1 + d_1)$ div $2 <= (n + n + 1)$ div $2 = n$

   (b) $1 \le m$
   since $(g_1 + d_1)$ div $2 > 2$ div $2 = 1$

2. We symbolically execute the three conditional statements. We detail the first one: $a[m] < x$.

   the execution of the iteration terminates with

   $g = m + 1$, $d = d_1$, $b = b_1$.

   The invariant is satisfied:

   - $\forall i : 1 \le i \le g - 1 : a[i] < x$ because
   $a[m] < x$ and $\forall i : 1 \le i \le m - 1 : a[i] \le a[i+1]$ (from **(1)**,**(6)**)
   and so, $\forall i : 1 \le i \le m : a[i] < x$ with $m = g - 1$.
   - $1 \le g \le d$ because
   $$\begin{array}{rcll} g & = & m+1 & \\ & = & (g_1+d_1) \text{ div } 2+1 & (m = (g_1+d_1) \text{ div } 2) \\ & < & d_1+1 & (g_1 < d_1) \\ & = & d+1 & (d = d_1) \end{array}$$
   and
   $$\begin{array}{rcl} g & = & m+1 \\ & = & (g_1+d_1) \text{ div } 2+1 \\ & \ge & 1. \end{array}$$
   - The other parts of the invariant are satisfied because $d = d_0$ and $b = b_0$.

Finally, for this example, choosing an appropriate variant is not so simple: the decreasing integer function should probably take into account the size of the middle part of the array because this part tends to an empty part: in mathematical terms, $d - g$ is decreasing to 0. But is it sufficient? There is a second condition for terminating the loop: when the element x is found. The iteration consists in changing the value of b from $false$ to $true$. As the variant must be an integer function, we can, for example, "convert" $false$

and *true* to 0 and 1. One of the possible variants is if $b = true$ then $d - g - 1$ else $d - g$.
This variant is always positive, because

- if $b = true$, then $g < d$ and so $d - g > 0$ and $d - g - 1 \geq 0$;
- otherwise, $g \leq d$, and so $d - g \geq 0$

Notice that before the iteration, $g < d$ and $b = false$, and so, $d - g > 0$.

It is important to observe that many students would have written `g := m` instead of `g := m+1`. The three Hoare propositions are correct but we cannot find a suitable variant simply because the algorithm does not terminate. In fact, when the subarray $a[g..d-1]$ contains only one element, we can have $g = (g + d)$ div 2, if we execute the statement `g := m`, it does not modify g and the halting condition is never reached.

This algorithm is an interesting example involving many representative aspects of the method. Using a rigorous method is essential and without any invariant picture, we can definitly not be sure to construct a correct algorithm. We can try to convince students by giving them the same exercise with another index convention choice. Indeed, for this exercise, the invariant helps for

- choosing a well-defined middle element of a subarray,
- correctly updating the index g and d in such a way that the algorithm terminates,
- correctly choosing the halting condition.

The proof of the correctness has shown that formalising the invariant is not always easy. There are some risks of inconsistency between formal and pictured versions, especially when students are not used to write assertions. Providing an appropriate variant is also not simple but it is useful.

2.1.4 The Insertion Sort Algorithm

The programming method that we are presenting tends to reduce the difficulties by dividing the problem and solving small subproblems instead of solving globally the main problem.
We now introduce a more general idea of decomposition into subproblems. A problem can consider some *complex operations* as subproblems that will be solved independently from each other. To discuss this decomposition into subproblems, we develop the insertion sort algorithm.

Constructing a correct algorithm by decomposition into subproblems, and using the invariant method As usual, we begin with the specification:

Precondition The array (denoted by a) just needs to be initialised.

Postcondition a is a permutation of the initial a (denoted a_0) and is sorted (increasing).

To have in mind the insertion sort algorithm, we have a look at an example: at each iteration the "next" element is inserted at the right place in the sorted subarray.

- At the beginning,

$$\mathtt{a} : \quad \underbrace{}\begin{array}{|c|c|c|c|}\hline 2 & 1 & 4 & 3 \\ \hline \end{array}$$

- After inserting 2 in the sorted subarray,

$$\mathtt{a} : \quad \underbrace{\begin{array}{|c|}\hline 2 \\ \hline \end{array}}_{\text{sorted}}\begin{array}{|c|c|c|}\hline 1 & 4 & 3 \\ \hline \end{array}$$

- After inserting 1 in the sorted subarray,

$$\mathtt{a} : \quad \underbrace{\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array}}_{\text{sorted}}\begin{array}{|c|c|}\hline 4 & 3 \\ \hline \end{array}$$

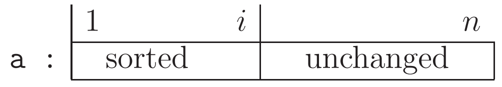
- After inserting 4 in the sorted subarray,

$$\mathtt{a} : \quad \underbrace{\begin{array}{|c|c|c|}\hline 1 & 2 & 4 \\ \hline \end{array}}_{\text{sorted}}\begin{array}{|c|}\hline 3 \\ \hline \end{array}$$

- After inserting 3 in the sorted subarray,

$$\mathtt{a} : \quad \underbrace{\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline \end{array}}_{\text{sorted array}}$$

The loop invariant can be choosen:



$0 \leq i \leq n$
a is a permutation of a_0
$a[1..i]$ is sorted
$a[i+1..n]$ is unchanged

Again, keeping the same index convention along the construction of the algorithm is essential.
In the example, the first iteration seems useless, and restricting the invariant to $1 \leq i \leq n$ instead of $0 \leq i \leq n$ could seem appropriate, it is not when the array is empty ($n = 0$) and it is better to have a more global approach.

So, the initialisation statement is

```
i := 0
```

The array is completely sorted when $i = n$, so the halting condition is `i=n`.
No closure is needed and the iteration consists in inserting $a[i+1]$ at the right place in $a[1..i+1]$, and then incrementing i. A subproblem `SP` is responsible for the insertion of $a[i+1]$.

```
SP ; i := i + 1
```

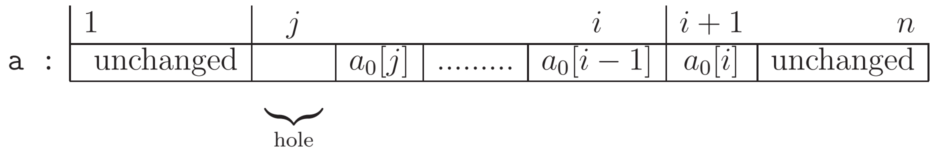
We specify the subproblem:

Precondition: $0 \leq i < n$ and $a[1..i]$ is sorted (increasing)

Postcondition: $a[1..i+1]$ is sorted (increasing) and a is a permutation of a_0 and $a[i+2..n]$ is unchanged

According to this specification, the subproblem can be constructed.

To insert $a[i+1]$, we have to find the correct position, and shift the left values of that position to the right. For complexity reasons, we make two operations at the same time: we look at a place $a[j-1]$; if it is strictly greater than $a[i+1]$, we shift it, we finish when $a[j-1]$ is smaller and we can place $a[i+1]$ in the $a[j]$ place. The loop invariant follows.



i is unchanged
$1 \leq j \leq i+1$
$a[i+2..n]$ is unchanged
$a[1..j]$ is unchanged
$x = a_0[i+1]$
$\forall k : j+1 \leq k \leq i+1 : a[k] > x$
$\forall k : j \leq k \leq i : a[k+1] = a_0[k]$

Before the first iteration, nothing has been changed in the array and x contains $a[i+1]$; we can derive the initialisation statements:

```
j := i + 1; x := a[j]
```

The iteration consists in shifting an element to the right and updating j:

```
a[j] := a[j - 1]; j := j - 1
```

The loop terminates when we are at the beginning of the array or when $a[j]$ is the right place for x, i.e., $a[j-1] \leq x$; the halting condition is

```
j = 1 || a[j - 1] <= x
```

The Boolean operator `||` stands for a disjunction evaluation from left to right, we evaluate the second term only if the first term is not satisfied. A strict evaluation would generate an out of bound exception because if $j = 1$ then $a[j-1]$ is not defined.

To satisfy the postcondition, a statement (the closure) is needed after leaving the loop; we set x at its place:

```
a[j] := x
```

and so, the subarray $a[1..i+1]$ is sorted and a is a permutation of a_0.

A final version is depicted in Figure 2.3. In postcondition, *"a is increasing"* is formalised and permut(a,a_0) stands for *"a is a permutation of* a_0*"*.

Proving the correctness of the main algorithm We suppose that the subproblem is correct according to its specifications (the proof is left to the reader) and we prove the correctness of the main problem. We first verify that the proposition $\{Inv$ and not $H\}$ $Iter$ $\{Inv\}$ holds. This contains a subproblem call.

Let a_1, i_1 be the initial values of a, i;
by hypothesis,
$0 \leq i_1 < n$ and $\forall j : 1 \leq j < i_1 : a_1[j] \leq a_1[j+1]$
and a_1 is a permutation of a_0 and $a_1[i_1+1..n]$ is unchanged.

- Declarations:
  const n ;
  tab a : array [1..n] of integer; (input, output)
  var i , j , x : integer; (auxiliary variable)
- Pre: a initialised and $n \geq 0$
  Post: $\forall j : 1 \leq j < n : a[j] \leq a[j+1]$ and permut(a,a_0)
-
```
i := 0;
while(i <> n)
   j := i + 1;
   x := a[i + 1];
   while(j <> 1 && a[j - 1] > x)
       a[j] := a[j - 1];
       j := j - 1
   end
   a[j] := x;
   i := i + 1
end
```

Figure 2.3: The insertion sort algorithm

First, according to this hypothesis, we verify that the precondition of the subproblem is satisfied with the values i_1 and a_1. It is satisfied.

Next, let us verify that the invariant holds after the iteration.
Indeed, the execution terminates with
$a[i_1+2..n]$ unchanged with regard to $a_1[i_1+2..n]$ and $\forall k : 1 \leq k \leq i_1 : a[k] \leq [k+1]$ and a is a permutation of a_1 and $i = i_1 + 1$. It is important to notice that we do not pay attention to the implementation of the subproblem, we just need its specification.
The final values a and i satisfy the invariant:

- $0 \leq i \leq n$ because $i = i_1 + 1 < n + 1$;
- giving that unchanged is a reflexive and transitive relation, and that if the relation is true for $a[x..y]$, the relation is true for any subarray of $a[x..y]$,

  we can argue that $a[i+1, ..n]$ is unchanged because

  - $a[i_1 + 2..n]$ is unchanged with regard to $a_1[i_1 + 2..n]$
  - and $a_1[i_1 + 1..n]$ is unchanged compared to $a_0[i_1 + 1..n]$ and so $a_1[i_0 + 2..n]$ is unchanged compared to $a_0[i_1 + 2..n]$;
  - and $i_1 = i - 1$.

- $(\forall k : 1 \leq k < i : a[k] \leq a[k+1])$ because $(\forall k : 1 \leq k \leq i_1 : a[k] \leq a[k+1])$ and $i_1 = i - 1$;
- giving that permut is a reflexive and transitive relation,

  permut(a,a_0) because permut(a, a_1) and permut(a_1, a_0).

Now, let us verify that the invariant holds after the initialisation: after the initialisation, $i = 0$.

- $0 \leq i \leq n$ holds if $i = 0$;
- a is a permutation of a_0 because $a = a_0$ after the initialisation;
- $a[1..0]$ is sorted
- $a[1..n]$ is unchanged because a is not modified upto now.

We verify the proposition $\{Inv$ and $H\}$ *Clot* $\{Post\}$:
by hypothesis,
$i = n$ and $\forall j : 1 \leq j < i : a_1[j] \leq a_1[j+1]$
and a is a permutation of a_0 and $a[i+1, ..n]$ is unchanged.
This is equivalent to the postcondition.

To guarantee total correctness, we use the variant $n - i$. This function is positive because $0 \leq i \leq n$ and is decreasing at every iteration, because i increases. So, our algorithm terminates.

Looking at the final version of this sorting algorithm, we can easily be convinced that we needed to use a rigorous method to correctly solve this problem. It is not by trial and error that we can get this short, clear and correct program. The decomposition into subproblems clearly helps for the construction and the verification of the algorithm. For the construction, one can observe that, to avoid too many formal sentences, we use precise shortcuts to express known notions as a permutation or the fact that a subarray is unchanged.

2.1.5 An Algorithm to find the Next Permutation

As last example, we consider a more complex algorithm whose construction and verification need a decomposition into four subproblems. Given a permutation of the natural numbers $1, \ldots, n$, the problem is to write an algorithm that finds the *next permutation* according to the lexicographic ordering (denoted by $\preceq$). For example, the next permutation of

$$\begin{array}{r|c|c|c|c|c|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
\texttt{a:} & 1 & 2 & 3 & 5 & 4 \\
\hline
\end{array}$$

is

$$\begin{array}{r|c|c|c|c|c|} & 1 & 2 & 3 & 4 & 5 \\ \cline{2-6} \texttt{a :} & 1 & 2 & 4 & 3 & 5 \\ \cline{2-6} \end{array}$$

Informally, by means of the example, this problem can be decomposed as follows:

1. First (SP1), we compute the largest index i such that $a[i] < a[i+1]$.

$$\begin{array}{r|c|c|c|c|c|} & 1 & 2 & \texttt{i} & 4 & 5 \\ \cline{2-6} \texttt{a :} & 1 & 2 & 3 & 5 & 4 \\ \cline{2-6} \end{array}$$

2. Then (SP2), we determine j, which is the index of the smallest element on the right side of $a[i]$ that is larger than $a[i]$.

$$\begin{array}{r|c|c|c|c|c|} & 1 & 2 & 3 & 4 & \texttt{j} \\ \cline{2-6} \texttt{a :} & 1 & 2 & 3 & 5 & 4 \\ \cline{2-6} \end{array}$$

3. Next, we exchange $a[i]$ and $a[j]$

$$\begin{array}{r|c|c|c|c|c|} & 1 & 2 & \texttt{i} & 4 & \texttt{j} \\ \cline{2-6} \texttt{a :} & 1 & 2 & \mathbf{4} & 5 & \mathbf{3} \\ \cline{2-6} \end{array}$$

4. Finally (SP3), we reverse the order of the elements in the subarray $a[i+1..n]$

$$\begin{array}{r|c|c|c|c|c|} & 1 & 2 & \texttt{i} & 4 & 5 \\ \cline{2-6} \texttt{a :} & 1 & 2 & 4 & \mathbf{3} & \mathbf{5} \\ \cline{2-6} \end{array}$$

   a is the next permutation.

To construct the algorithm, we must specify the three subproblems in a more formal way than in the explanation above.

SP1: The largest index such that $a[i] < a[i+1]$ means that $(\forall\ k : i+1 \leq k \leq n-1 : a[k] \geq a[k+1])$ and $a[i] < a[i+1]$; but, there is also the case where there is no i such $a[i] < a[i+1]$, i.e., when $i = 0$. The complete formal specification is:

Pre: a initialised and $n > 0$
Post: a unchanged and
$(i = 0$ and $(\forall\ k : 1 \leq k \leq n-1 : a[k] \geq a[k+1]))$
or
$(0 < i < n$ and $(\forall\ k : i+1 \leq k \leq n-1 : a[k] \geq a[k+1])$ and $a[i] < a[i+1])$

SP2: Expressing the postcondition of SP2 is not trivial: it must express that $a[j]$ is the least element greater than $a[i]$, on its right. We might write an assertion involving the minimum notion but it is simpler (and more efficient for the design of the algorithm) to consider in the precondition states that the subarray $a[i+1..n]$ is decreasing and that $a[i] < a[i+1]$. In this way, it is sufficient to write that $a[j] > a[i]$ and $a[j+1] \leq a[i]$ if $j < n$, but the case $j = n$ needs a special treatment, since there is no element $a[n+1]$. The complete formal specification is:

Pre: a initialised and $0 < i < n$
and $(\forall\ k : i+1 \leq k \leq n-1 : a[k] \geq a[k+1])$ and $a[i] < a[i+1]$
Post: a, i unchanged and $i < j \leq n$ and $a[j] > a[i]$ and
$(j < n$ and $a[j+1] \leq a[i])$ or $j = n$

SP3: For this subproblem, we have to be careful about the indexes computing to express the reversing of the subarray $a[i+1..n]$.

The symmetric element of $a[(i+1)+k]$ is $a[n-k]$ with $0 \leq k \leq n-(i+1)$ or else,

the symmetric element of $a[k]$ is $a[n-(k-(i+1))]$ with $i+1 \leq k \leq n$. The complete formal specification is:

Pre: a initialised and $0 < i < n$
Post: i inchanged and $a[1..i]$ unchanged
and $(\forall\ k : i+1 \leq k \leq n : a[k] = a_0[n-k+i+1])$

The main difficulty of the design of this algorithm is this translation of the informal specifications into pictures or into mathematical formulas. After that, the construction of each subproblem does not involve a lot of difficulties. We notice that the natural trend is to restrict specifications to the context of the main problem. For example, in subproblem preconditions, it may be more natural to restrict a to a permutation of $1..n$; however, it is better to construct more general subproblems. We can observe that the postcondition of SP1 is larger when the precondition does not restrict the values of the array a (we have non strict inequalities that we would not have with the restricted version). However, for SP2, we keep a restriction on a in precondition to make the design of its algorithm easier.

Anyway, the elaboration of the code of the main algorithm is very simple when the subproblems are solved. It does not involve an iteration. Hence no invariant is needed. Let us first clarify the specifications: in precondition, a is a permutation of $1..n$; in postcondition, if b is *true* then a is the next permutation of a_0; if b is *false*, then a is the last permutation of $1..n$. As a sequence of statements, we first execute SP1: precondition of SP1 is satisfied because a is well initialised in precondition of the main algorithm. Then,

- if $i = 0$, it means that a is the last permutation of $1..n$; we assign false to b;

- if $i > 0$,
  - we execute SP2: precondition of SP2 is satisfied because it corresponds to the postcondition of SP1 when $i > 0$.
    At this moment, i and j are determined, and $0 < i < j \leq n$.
  - We can permute $a[i]$ and $a[j]$, these elements are well defined.
  - Finally, we execute SP3,
    this subproblem only requires that $0 < i < n$, which is guaranteed because of SP1 postcondition and because i is unchanged in the other subproblems.
  - The resulting array is the next permutation of a_0; we assign true to b.

We provide the complete version of this algorithm in Figure 2.4. One can observe that the formal postcondition of the main algorithm is not simple. The formalisation of *"a is the next permutation of a_0"* needs a universal quantification over an array variable: to express it, we say that among all the permutations of $1 \ldots n$ which are greater than a_0, a is the smallest; to express that a is the last permutation, we say that all the permutations of $1 \ldots n$ are smaller.

Writing correct formal specifications for the subproblems is needed to efficiently construct the subproblems, it is also the key issue to prove the correctness of the complete algorithm in a compositional way.

Proving the correctness of the algorithm by symbolic execution By symbolic execution we propagate assertions. We execute the main algorithm and we determine the strongest assertion at each program point. Before each subproblem call, the subproblem precondition must hold; after each subproblem call, the subproblem postcondition is satified, and the assertion of the previous program point still holds when the variables are unchanged. When some variables are changed, these variables considered in the previous program point are renamed, and the previous assertion holds with these renamed variables.

Let A_0 be the initial assertion. We analyse the sequence of statements related to the case where a next permutation ($b = true$) exists. We write a_0 for the initial values of a; by hypothesis, a_0 is a permutation of $1..n$ (we note this assertion by perm1N(a_0)).

$A_0 \equiv \{\text{perm1N}(a_0)\}$

```
SP1;
```

- Declarations:
  const n;
  var a: array $[1..n]$ of integer; (input, output)
  var b: Boolean; (output)
  var i, j, tmp: integer; (auxiliary variable)
- Pre: n is the array size, a is a permutation of $[1..n]$
  Post:
  $(b \Rightarrow (\text{permut}(a, a_0)$ and $a \succ a_0$
  and $(\forall\ c[1..n] : (\text{permut}(a_0, c)$ and $c \succ a_0) \Rightarrow c \succeq a)))$
  and
  $(\neg b \Rightarrow (a$ unchanged and $(\forall\ c[1..n] : \text{permut}(a, c)\ \Rightarrow c \preceq a)))$
-
```
i := n - 1;
while (( i > 0) && (a[i] >= a[i + 1]))  i := i-1 end ;
if (i > 0) then
     j := i + 1 ;
     while ((j <> n) && (a[j + 1] >  a[i])) j := j+1 end ;
     tmp := a[i] ; a[i] := a[j] ; a[j] := tmp ;
     l := i ;
     while (l <  i + (n - i) div 2)
           l := l + 1 ;
           tmp := a[l] ;
           a[l] : = a[n - (l - (i + 1))] ;
           a[n - (l - (i + 1))] := tmp
     end ;
     b := true
else b := false
end
```

Figure 2.4: Algorithm to find the next permutation

$$A_1 \equiv \left\{ \begin{array}{l} A_0 \text{ and} \\ (i = 0 \text{ and } (\forall\ k : 1 \le k \le n-1 : a_0[k] \ge a_0[k+1])) \\ \text{or} \\ (0 < i < n \text{ and} \\ (\forall k : i+1 \le k \le n-1 : a_0[k] \ge a_0[k+1]) \\ \text{and } a_0[i] < a_0[i+1]) \end{array} \right\}$$

`if (i>0)   SP2;`

$$A_2 \equiv \left\{ \begin{array}{l} A_1 \text{ and } 0 < i < j \le n \text{ and} \\ a_0[j] > a_0[i] \text{ and} \\ ((j < n \text{ and } a_0[j+1] \le a_0[i]) \\ \text{or } j = n) \end{array} \right\}$$

```
temp := a[i];
a[i] := a[j];
```

```
            a[j] := temp;
```

$$A_3 \equiv \left\{ \begin{array}{l} A_2 \text{ and} \\ a_0[1..n] = a[1..n] \text{ unless for } a[i] = a_0[j] \text{ and } a[j] = a_0[i] \end{array} \right\}$$

```
            SP3;
```

$$A_4 \equiv \left\{ \begin{array}{l} \text{Let } a_1[1..n] \text{ be the values of } a \text{ at the previous state :} \\ A_2 \text{ and } a_0[1..n] = a_1[1..n] \text{ unless for } a_1[i] = a_0[j] \text{ and } a_1[j] = a_0[i] \\ \text{and } a[1..i] = a_1[1..i] \text{ and} \\ (\forall k : i+1 \leq k \leq n : a[k] = a_1[n-k+i+1]) \end{array} \right\}$$

```
            b := true
```

$A_5 \equiv \{A_4 \text{ and } b = \text{true}\}$

```
else        b := false
```

$A_6 \equiv \{A_0 \text{ and } (\forall\ k : 1 \leq k \leq n-1 : a[k] \geq a[k+1])) \text{ and } b = \text{false}\}$

```
end
```

We first verify that the statements are well defined; at each subproblem call, its precondition holds:

- the SP1 precondition holds because in the main algorithm precondition, a is initialised;
- the SP2 precondition holds because, SP2 is called only when $i > 0$, and (A_1 and $i > 0$) implies the SP2 precondition;
- the SP3 precondition holds because, SP3 is called only when $i > 0$, and a is initialised in the main algorithm precondition.

Besides, the sequence

```
            temp := a[i];
            a[i] := a[j];
            a[j] := temp;
```

is well defined because, before the execution, $0 < i < j \leq n$.

Supposing that the subproblems are correct according to their specifications, A_0 is modified after every statement until it reaches the A_5 assertion, if a next permutation exists. Let us prove that the algorithm actually computes the next permutation, i.e., that A_5 expresses the fact that a is the next permutation of a_0. We leave to the reader the proof of that A_6 expresses that a is the last permutation. The proposition that we prove is the following:

$$\boxed{\begin{array}{l}
\textbf{(1)}\ \text{perm1N}(a_0),\\
\textbf{(2)}\ 0 < i < n,\\
\textbf{(3)}\ (\forall k : i+1 \leq k \leq n-1 : a_0[k] \geq a_0[k+1]),\\
\textbf{(4)}\ a_0[i] < a_0[i+1],\\
\textbf{(5)}\ i < j,\\
\textbf{(6)}\ a_0[j] > a_0[i],\\
\textbf{(7)}\ ((j < n \text{ and } a_0[j+1] \leq a_0[i]) \text{ or } j = n),\\
\textbf{(8)}\ \forall k : 1 \leq k \leq n : k \neq i, j \Rightarrow a_1[k] = a_0[k],\\
\textbf{(9)}\ a_1[i] = a_0[j],\\
\textbf{(10)}\ a_1[j] = a_0[i],\\
\textbf{(11)}\ \forall k : 1 \leq k \leq i : a[k] = a_1[k],\\
\textbf{(12)}\ (\forall k : i+1 \leq k \leq n : a[k] = a_1[n-k+i+1])\\
\Rightarrow\\
\textbf{(13)}\ \text{permut}(a, a_0),\\
a \succ a_0,\\
(\forall c[1..n] : (\text{permut}(c, a_0) \text{ and } c \succ a_0 \Rightarrow c \succeq a))
\end{array}}$$

- permut(a, a_0) because
  1. permut(a_0, a_1) because, according to (**8, 9, 10**), we can find a permutation f of $1..n$, such that $\forall k : 1 \leq k \leq n : a_1[k] = a_0[f(k)]$.
     Permutation f is defined as follows:
     $\forall k : 1 \leq k \leq n : (k \neq i, j \Rightarrow f(k) = k)$ and $f(i) = j$ and $f(j) = i$.
  2. permut(a, a_1) because (**11**) and **(12)** highlight a permutation f of $1..n$, such that
     $\forall k : 1 \leq k \leq i : f(k) = k$ and $\forall k : i+1 \leq k \leq n : f(k) = n-k+i+1$
     and so, by definition, a is a permutation of a_1.

  If f_1 and f_2 are permutations of $1..n$ then $f_1 \circ f_2$ is a permutation of $1..n$, and so, permut$(a,\ a_0)$ is true.

- $a \succ a_0$:

  we prove that $\exists k : 1 \leq k \leq n : (\forall s : 1 \leq s < k : a[s] = a_0[s])$ and $a[k] > a_0[k]$.

  We simply instantiate k with i:

  $(\forall s : 1 \leq s < i : a[s] = a_1[s] = a_0[s])$ by **(8, 11)**

  and $a[i] =_{(11)} a_1[i] =_{(9)} a_0[j] >_{(6)} a_0[i]$.

- let c be an array such that permut(c, a_0) **(14)** and $c \succ a_0$.

  $c \succ a_0$ means that

(15) $(\exists k : 1 \leq k \leq n :$

(16) $(\forall s : 1 \leq s < k : a_0[s] = c[s])$,

(17) $c[k] > a_0[k])$.

Suppose by contradiction that $c \prec a$ **(18)**.

We instantiate k satisfying **(15)**

- assume $k > i$:
  (14) implies that if **(16, 17)**, $c[k] \in a_0[k+1..n]$, i.e.,
  (19) $(\exists r : k+1 \leq r \leq n : a_0[r] = c[k] >_{\mathbf{17}} a_0[k])$.
  But, giving that $k > i$ and **(3)**,
  $(\forall r : k \leq r \leq n-1 : a_0[r] \geq a_0[r+1])$;
  and so, $(\forall r : k+1 \leq r \leq n : a_0[k] \geq a_0[r])$
  which is inconsistent with **(19)**.
- assume $k < i$:
  $(\forall s : 1 \leq s < k : a_0[s] = c[s] = a[s])$ by **(8, 11, 16)**
  and $c[k] >_{(\mathbf{17})} a_0[k] =_{(\mathbf{8},\mathbf{11})} a[k]$
  and so, $c \succ a$ contrary to **(18)**.
- assume $k = i$:
  $(\forall s : 1 \leq s < i : a_0[s] = c[s] = a[s])$ by **(8, 11, 16)**
  $c[i] >_{(\mathbf{17})} a_0[i]$ and $a[i] =_{(\mathbf{9},\mathbf{11})} a_0[j] >_{(\mathbf{6})} a_0[i]$;
  (18) means that
  (20) $(\exists l : 1 \leq l \leq n :$
  (21) $(\forall s : 1 \leq s < l : a[s] = c[s])$,
  (22) $c[l] < a[l])$.
  We instantiate l satisfying **(20)**
  - ∗ assume $l = i$:
    (14, 16, 17) implies that $c[i] \in a_0[i+1..n]$
    - · if $c[i] \in a_0[i+1..j-1]$ then $c[i] \geq_{(\mathbf{3})} a_0[j] =_{(\mathbf{9},\mathbf{11})} a[i]$ contrary to **(22)**;
    - · if $c[i] \in a_0[j+1..n]$ then $c[i] \leq_{(\mathbf{3})} a_0[j] <_{(\mathbf{6})} a_0[i]$ which is contrary to **(17)**;
    - · if $c[i] = a_0[j]$ then $c[i] = a[i]$ and **(22)** is not satified.
  - ∗ assume $l < i$:
    $(\forall s : 1 \leq s < l : a[s] = c[s] = a_0[s])$ by **(8, 11, 16)**
    $c[l] <_{(\mathbf{20})} a[l] =_{(\mathbf{8},\mathbf{11})} a_0[l]$;
    so, by definition, $c \prec a_0$, inconsistent with **(15)**.

- $*$ assume $l > i$:
  $(\forall s : l \leq s \leq n : a[s] \leq a[s+1])$ (*)
  (13, 15, 21, 22) implies that $c[l] \in a[l+1..n]$ and so, $c[l] \geq a[l]$
  which is contrary to **(22)**.
  (*)Indeed,
  - $\cdot$ **(23)** $(\forall k : i+1 \leq k \leq n-1 : a_1[k] \geq a_1[k+1])$
    from **(3)** and

    when $j < n$,
$$\begin{aligned} a_1[j] &=_{(\mathbf{10})} a_0[i] \\ &\geq_{(\mathbf{7})} a_0[j+1] \\ &=_{(\mathbf{8})} a_1[j+1] \end{aligned}$$
    and when $j > i+2$,
$$\begin{aligned} a_1[j] &=_{(\mathbf{10})} a_0[i] \\ &\leq_{(\mathbf{6},\mathbf{3})} a_0[j-1] \\ &=_{(\mathbf{8})} a_1[j-1] \end{aligned}$$
  - $\cdot$ $(\forall k : i+1 \leq k \leq n-1 : a[k+1] > a[k])$ because
    $\forall k : i+1 \leq k \leq n-1 :$
$$\begin{aligned} a[n-k+i+1] &=_{\mathbf{12}} a_1[k] \\ &\geq_{\mathbf{23}} a_1[k+1] \\ &=_{\mathbf{12}} a[n-(k+1)+i+1] \\ &= a[n-k+i] \end{aligned}$$
    with $i+1 \leq n-k+i \leq n-1$,

  and so, the conclusion holds.

As in the previous example, this exercise highlights the advantages of the decomposition into subproblems. It also shows that it is not simple to prove that the succession of these subproblems results in a next permutation in the lexicographic order even if the person who chooses this decomposition can feel (through examples) it should work. The formalisation of the postcondition and the mathematical proof of the correctness are particularly tedious. In fact, the programming method converts the complexity of elaborating a program to a purely mathematical reasoning approach, which is not always so obvious.

2.2 Pedagogical Findings

Ten years ago, the invariant method was assiduously taught from the introductory programming course [37]. The main purpose of an introductory course was the algorithmic learning; constructing programs with rigor and formal specifications was considered very important. Today, efforts are often mainly deployed on teaching oriented-object concepts as exemplified by the JAVA language. Learning algorithmics with a structured method is no longer a priority among the courses. One reason probably is that students have a hard time to understand the method, another reason may be the general discouragement from teachers partially due to the observation that the method will never be completely supported by an automatic tool.

In our view, we consider that, anyway, algorithmics learning using this construction method is fundamental in computer science. There are many concepts, such as decomposition into subproblems, looking for simplicity, reasoning, requirements, etc. that we introduce through this learning and that are very important notions overall in computer science. So, in our courses, we continue to give major role to algorithmics and to the structured programming method, we continue to teach the invariant method to help constructing (simple) programs more systematically, but we do it very informally in the introductory course [38]. Then, in a more advanced course [39], we review the method more formally, and we expect the students to get a deeper understanding of it.

The first section explains the reasons why such a method is appropriate in a universitary cursus. Section 2.2.2 describes the difficulties that students meet when they learn this method; finally, in Section 2.2.3, we propose to overcome these difficulties by using verification tools.

2.2.1 Pedagogical Motivations

From the first approach to programming, learning the use of a structured method for building an algorithm is crucial. The method that we propose has the big advantage of **constructing correct programs**. This also leads the programmer to write simple and readable programs. Unhappily, the method is not simple to learn and it requires a lot of experiments and imagination from the beginner. Through this learning, students learn many other important notions, they **decompose a problem** into subproblems, they understand the role of a **complete specification**, they learn to be **rigorous** and gain a better **abstraction level**. The systematic decomposition into subproblems is the spirit of computer science, teaching this good reflex with rigour is essential even to the beginners. Also, writing precise specifications and being rigorous should be the practice of any programmer; and a good computer scientist will always need to have a good abstraction capability.

Concretely, we are conscious that, with novice students, it is difficult to focus on mathematical formalisation of assertions, and so, we just construct algorithms by representing the specifications and the invariant with pictures; representing these pictures requires abstraction and rigor that cannot be minimised. In a more advanced course, the same concepts are reviewed with more background. We learn to formalise the pictures and to adopt a mathematical reasoning on algorithms; the abstraction level is higher and the student can be conscious that we are able to prove the correctness of programs.

2.2.2 Pedagogical Difficulties

Studying computer science requires a good level of abstraction: an algorithm is abstract, a specification is more abstract, and formal specification is even more abstract. It is not amazing that this method is difficult to learn. The problem is that most students never feel the advantages of the method, they feel it as a constraint and they do not understand the role of the invariant in the constructive approach. The main reasons are:

- Thinking in terms of situations is not intuitive.

  For many novice students, the difference between statement and assertion is not obvious, and it takes a long time to feel correctly the meaning of an assertion and an invariant.

  Besides, after having understood these notions, the student has to think in terms of situations and has to discover an invariant, which is not intuitive for a beginner.

  Students have some difficulties to understand deeply the meaning of a decomposition into subproblems: to construct or verify a program calling a subproblem, we do not need the implementation of the subproblem, the specifications are used instead; the subproblem is constructed or verified independently. For a loop construction, each Hoare condition is completely independent from the others. For example, we do not need the initialisation statements when we construct or verify the loop transformation. In fact, many students do not understand what is a complete invariant, even for simple examples.

  Facing with these difficulties, most students do not use the method, they prefer to solve globally the problem by trial and error and finally they write a posteriori a kind of invariant (often incorrect or incomplete) for the teacher.

- Lack of mathematical maturity.

  Abstraction or formalisation, these two mathematical requirements are seldom gained by our students and we cannot overcome this general problem by our own.

  And yet, the splendor of verification approach of this method can be successful only if students are able to write specifications (and invariants) that are both sound and complete. If the method is not supported by a formal specification language, it is very hard to explain what a good specification (or invariant) is.

We notice that many students, even advanced students have some difficulties to express pictured assertions formally.

Besides, formalising assertions may sometimes be more complex than writing the algorithm. Most students do not find satisfaction, the magic of the proof of a program correctness does not seem to interest them.

Next to these problems of abstraction, there is a lack of motivation. We have a few possible explanations:

- It misses an authority argument.

  This method is too often viewed as an isolated method without any link with the other courses. For, example, if we are alone to insist on complete and formal specifications, the impact of this learning will not have the same effect than if every teacher whould send the same message. This lack of coherence may demotivate students.

- Students think that the design of an algorithm together with its specifications using this method takes too much time and is too complex. It fact, it is very difficult to convince students that this method allows them to write simple and correct algorithms and to gain time.

- Reasoning on paper does not motivate students; they are used to write their programs directly on the computer without any explicit reasoning.

2.2.3 Our Proposal to Overcome the Difficulties

The structured programming method encourages a rigorous behaviour, teaches mathematical formalism, increases the abstraction capacity and so, we are convinced that this method is appropriate in a university cursus: we do not want to forget this method arguing that it is too complex for students, we wish to help them to overcome these pedagogical obstacles.

Since our students are often more motivated by using a computer than by reasoning on paper, we propose to stimulate them by using automatic verification tools. To be more precise, we would like to help our students by providing them an **appropriate** tool that enforces the use of the method: decomposition into subproblems, complete specification of the subproblems and loop invariants. Moreover, the automatic verification should provide an informative feedback to the students when they make specification or reasoning errors. Furthermore, we would like a tool that charms the students with the magic of the proofs of the algorithm correctness.

In the next chapter, we evaluate several automatic verification tools in our context of programming method learning. First, we try ESC/Java [23, 12, 16], a fully automatic tool that verifies Java programs and uses JML, a very expressive assertion language; the weakness of this tool is that it is neither complete (it gives false warnings) nor sound (it forgets errors). We analyse the ESC/Java behaviour when it verifies pedagogical algorithms from our teaching context. Then, we try SparkAda [3], which is partially automatic and verifies Ada programs; this tool is able to be complete and sound if the user has some expertise in interactive theorem prover. We observe the set of algorithms that can be verified automatically, without using the interactive theorem prover, and we evaluate the feedback given when errors occur; the weakness of this tool is that the interactive theorem proving requires expertise from the user; we evaluate this expertise level and we conclude about the adequacy of this tool in our programming courses. Finally, we use SMV [44] to verify our algorithms; this tool works on finite domains, and is complete and sound; we evaluate the difficulty level to translate our programs and assertions in the model checker language and we analyse the feedback of this tool.

Chapter 3

How about Existing Tools to Teach Structured Programming?

3.1 Extended Static Checker

ESC/Java developed at DEC/SRC [1] was a pioneering tool in the application of static analysis and verification technology to annotated Java programs [23]. It was the successor to the ESC/Modula-3 tool described in [16]. The basic project was studying the engineering practices of best programmers and it was developing a tool that improved the quality of programs, without changing their essential nature [16]. The programmers can annotate their programs with JML specifications, loop invariants, and intermediate assertions; other annotations can be associated to field declarations, method declarations and class declarations [7, 41]. According to these annotations, ESC/Java catches errors at compile time which are not ordinarily caught until runtime. Let us mention examples like array index bounds errors, nil dereferences, and deadlocks and race conditions in multi-threads programs. ESC/Java was not designed to be complete nor entirely sound, but claims to operate on full Java programs, keeping the complexity low for industrial-sized programs. The ESC/Java2 tool, an extension of ESC/Java, has continued that spirit, though some unsound aspects have been corrected [12].

Most applications of ESC/Java and ESC/Java2 are performed by the SoS group at the University of Nijmegen, with other members of the European

[1]Research laboratory created by Digital Equipment Corporation (DEC) in 1984, in Palo Alto, California.

VerifCard Project. They focus on the verification of Java Card programs [2]. For example, the decimal representation in Java for smart card, a problem that has been described and proved interactively in [6], and has been updated and re-verified with ESC/Java2 [12]. The GemPlus Electronic Purse case study has been verified by ESC/Java and then by ESC/Java2 [8, 12]. A major partial verification using ESC/Java2 is the verification of a vote counting system (KOA system) in 2004, for the Dutch Parliament [12].

Given the importance of Java in academia, given the expressivity of JML, given the ambition of ESC/Java2 and an attractive demonstration at MOVEP'04 winter school [41], we propose to evaluate the impact of such a tool to convince students of the utility of a formal method to construct and verify programs. We wonder if this tool motivates students for writing complete and formal specifications. In this section, we first detail a subset of the specification language JML. Then we present the verification technique used by ESC/Java, with the main role given to the theorem prover Simplify [15]; we explain where the tool loses completeness and soundness. Next, we give a general idea of the ESC/Java2 behaviour through some examples of algorithm verification. Finally, we conclude about the adequacy of ESC/Java2 in our programming learning context.

3.1.1 The Java Modeling Language

The Java Modeling Language (JML) is a large specification language. Although JML has been designed in an oriented object context, we only describe the JML subset that we need to construct and verify algorithm from our learning context. So, we focus on annotations related to statements: pre and post conditions, loop invariant, intermediate assertions. And we also consider annotations related to methods, saying for example which variables can be modified. Details about annotations related to classes and fields, principles of specification inheritance are described in [7, 41].

Specifications are included in the text of a Java program in special annotation comments, which start with an at-sign(`@`). JML uses a `requires` clause to specify the implementor's precondition, and an `ensures` clause to specify its postcondition. The keyword `loop_invariant` is used to express loop invariant, and `assert` permits to attach an assertion to any program point; `decreasing E` is used to check that `E` decreases but stays non negative at each iteration. The keywords `modifies` and `pure` are related to a method, `modifies` specifies the set of possibly modified variables, `pure` stands for methods that do not modify any data.

[2] Java Card is a technology that enables smart cards and other devices with limited memory to run small applications, called applets, that utilise Java technology.

The syntax of specifications follows Java closely. It excludes any operation that has side effects (such as the increment operator `++`). Allowed operations such as arithmetic and comparison operators, have the syntax and the semantics of Java. Some expressions use extensions to Java such as `\result` which represents the result of a method call, `\old(E)` which is a way of referring to the pre-state value of an expression `E`. Other operators are also available, such as implication `==>` and equivalence `<==>`. Besides, JML supports several kinds of quantifiers in assertions: a *universal quantifier* `\forall`, an *existential quantifier* `\exists`, *generalised quantifiers* `\sum`, `\product`, `\min` and `\max` and a *numeric quantifier* `\num_of`. For example, the following predicate uses a universal quantifier:

```
   (\forall int i; 0 <= i & i < g ; a[i] < x)
```

In a quantified formula, there is a declaration of a variable that is local to the quantifier (`int i`). This is followed by an optional range predicate: the range restricts the domain to which the quantifier applies (`0 <= i & i < g`). The body of the quantifier, `a[i] < x` in the above example, must be true for all the objects (or values) that satisfy the range predicate. `\sum`, `\product`, `\min` and `\max` return respectively the sum, the product, the minimum and the maximum of the value of their body when the quantified variables satisfy the given range expression; `\num_of` returns the number of values for quantified variables for which the range and the body predicate are true. Specification predicates may include method calls, if those methods are pure; tools supporting JML can check that the implementation of pure methods have no side-effect.

JML is associated to a number of tools, which are provided to address the various needs such as reading, writing and checking JML specifications. A way of checking JML specifications, is runtime assertion checking, i.e., simply running the Java code and testing for violations of JML assertions. Such runtime assertion checks are accomplished by using the JML compiler (*jmlc*). The unit testing tool *jmlunit* combines runtime assertion checks with unit testing [10]. More ambitious is ESC/Java2 which tries to statically verify that the code satisfies its specifications.

3.1.2 The ESC/Java2 Verification Technique

Given a Java program, ESC/Java2 automatically derives and checks a set of verification conditions corresponding to a defined class of errors. Furthermore, it also allows the programmer to record design decisions, and to guide the choice of verification conditions by annotating the program with specifications, assertions, loop invariants by means of JML clauses. Verification conditions are submitted to the automatic theorem prover Simplify [15].

Simplify's input is an arbitrary first-order formula, including universal and existential quantifications. Simplify handles propositional connectives by backtracking search and includes complete decision procedures for the theory of equality and for linear rational arithmetic; but it is not complete for the linear theory of integers nor for the theory of nonlinear multiplication. Simplify does not support mathematical induction. The semantics of McCarthy's functions for updating and accessing arrays are also pre-defined. The two important modules in the prover are the *E-graph* module and the *Simplex* module. Each module implements a method to detect contradiction. The *E-graph* module focuses on the theory of equality with uninterpreted function symbols; the *Simplex* module focuses on rational linear arithmetic. Besides, we need a satisfiability procedure for the combination of the theories. To achieve this, it is necessary for the two modules to cooperate, according to a protocol known as *equality sharing*, introduced by Nelson and Oppen [47]. Simplify handles quantifiers by pattern driven instantiation. For instance, `(\forall T t1, ...,tn; B)` is verified by selective instantiating the body `B` with substitutions for `t1,...,tn` determined by matching certain "triggering patterns" against a set of terms already under consideration. It matches up to equivalence in an E-graph, which detects relevant pattern instances. But in some cases, the triggering may be too restrictive, preventing Simplify from finding instances.

To analyse an algorithm with a loop, ESC/Java2 considers only those that execute at most one complete iteration. In this way, it does not need an invariant. The user can unroll a loop more times, by specifying the number of iterations ESC/Java2 should consider (with `-loop n`). If the argument of the option is $n.0$ where n in a non-negative integer literal, to check the program fragment

```
while (B) {S}
```

ESC/Java2 will consider n executions of

```
if (!B) break;
S
```

If the argument of the option is $n.5$, ESC/Java2 will consider n executions of

```
if (!B) break;
S
```

plus one additional execution of

```
if (!B) break;
```

The default behaviour of ESC/Java2 is given by `-loop 1.5`

In terms of verification, ESC/Java2 can give spurious warnings: this is due to the fact the verification conditions generated by ESC/Java2 are submitted to Simplify which is not complete. Then, the proofs that Simplify does not conclude lead to error messages. ESC/Java2 can miss real programming errors, i.e., it is unsound: this is partially due to a tradeoff between the efficiency of ESC/Java2 and its semantic completeness. The verification conditions generated may not be exactly a translation of the annotated program. In particular, ESC/Java2 does not consider all possible execution paths through a loop. Finally, other sources of unsoundness and uncompleteness exist such as search limits of Simplify and the fact that ESC/Java2 does not check arithmetic overflows.

3.1.3 Experimentation on Some Examples

3.1.3.1 The Binary Search Algorithm

First, we present a correct Java version of the algorithm, together with its JML specifications. One can observe the adequate expressivity and readability of JML for this exercise. This can be a source of motivation to be rigorous, complete and formal in writing the specifications.

```
/*@requires a!=null &&
            (\forall int i,j; 0 <= i & i < j & j < a.length ;
                                a[i] <= a[j]);

  @ensures  \result <==>
            (\exists int j ; 0 <=j & j < a.length ;
                               a[j] == x) ;
  @*/
public /*@ pure @*/ static boolean dicho(int[] a, int x)
{
   int n = a.length ;
   int m ;
   int g = 0 ;
   int d = n ;
   boolean b = false ;

 /*@loop_invariant 0 <= g & g <= d & d <= a.length ;
   @loop_invariant (\forall int i ; 0 <= i & i < g ;
                                     a[i] < x) ;
   @loop_invariant (\forall int i ; d <= i & i < a.length ;
                                     a[i] > x) ;
   @loop_invariant b ==>
                   (\exists int j ; 0 <= j & j < a.length ;
                                     a[j]==x) ;
   @*/

    while ((b == false) & (g != d)){
         m = (g + d) / 2 ;
          if (a[m] < x) g = m + 1;
          else if (a[m] > x) d = m;
          else b = true;
    }

    return b;
}
```

Let us observe the ESC/Java2 behaviour through the binary search algorithm verification. First we display its feedback when it proves the correctness; then we intentionally insert errors in code or specifications and we assess the pertinence of the generated counter-examples. We arbitrary choose to unroll the loop 4 times: we run `escj -loop 4 Dicho.java`

Feedback Information

No error occurs Here is the message of ESC/Java2:

```
ESC/Java version ESCJava-2.0a9
    [0.155 s 4395512 bytes]

Dicho ...
  Prover started:0.032 s 6380112 bytes
    [1.455 s 6015480 bytes]

Dicho: dicho(int[], int) ...
    [2.339 s 6378800 bytes]  passed

Dicho: Dicho() ...
    [0.052 s 6534184 bytes]  passed
  [3.849 s 6535064 bytes total]
```

The proof was successful.

An error in the initialisation Imagine `b` is set to true in the initialisation, the tool is able to detect that the invariant does not hold; the violated part of the invariant is displayed:

```
Dicho.java:24:
  Warning: Loop invariant possibly does not hold (LoopInv)
  while ((b == false) & (g != d)){
           ^
Associated declaration is "Dicho.java", line 23, col 10:
  @loop_invariant b ==> (\exists int j ; 0 <= j & j < a.length
                         ==> ( ...
                ^
Execution trace information:
  Reached top of loop after 0 iterations in "Dicho.java",
  line 24, col 6.
```

We can interpret that before the first iteration, i.e., after the initialisation, the last part of the invariant is violated. No concrete counter-example is displayed, i.e., no concrete values are given for each program variable to attest that the implication does not hold.

An error in the invariant Instead of line 2 in the invariant, the user writes:

```
@loop_invariant (\forall int i ; 0 <= i & i <= g ;
                                 a[i] < x) ;
```

the tool displays a similar message:

```
Dicho.java:21:
   Warning: Loop invariant possibly does not hold (LoopInv)
   while ((b == false) & (g != d)){
            ^
Associated declaration is "Dicho.java", line 18, col 10:
   @loop_invariant (\forall int i ; 0 <= i & i <= g ;
                    a[i] < x) ;
                   ^
Execution trace information:
   Reached top of loop after 0 iterations in "Dicho.java",
   line 21, col 6.
```

The invariant does not hold after the initialisation.

An out of bound error Instead of comparing `a[m]` to `x`,

we compare `a[m+1]`, the tool clearly displays the out of bound error:

```
Dicho.java:24:
   Warning: Array index possibly too large (IndexTooBig)
   if (a[m+1] < x) {g = m+1;
        ^
Execution trace information:
   Reached top of loop after 0 iterations in "Dicho.java",
   line 21, col 6.
```

Again ESC/Java2 does not display a concrete counter-example.

A loop that does not terminate ESC/Java2 is not able to detect by itself that a loop does not terminate. But, if the user provides a variant, it can guarantee termination.

The annotation to use is `//@decreasing E;` where `E` is an arithmetic expression.

To express it in this algorithm, we propose to declare a new integer variable `y` initialised and modified so that the implementation satisfies

`//@loop_invariant y == 1 <==> b && y == 0 <==> !b ;`.
The adapted implementation stands in Annex A.1.1. When we write g := m ; instead of g := m + 1 ; in the iteration, ESC/Java2 verifies that the variant is positive and detects that the variant does not decrease:

```
Dicho.java:24:
   Warning: Loop variant function possibly not decreased
   while ((b == false) & (g < d)){
         ^
Associated declaration is "Dicho.java", line 22, col 10:
    //@decreasing d-g-y;
       ^
Execution trace information:
   Reached top of loop after 0 iterations in "Dicho.java",
   line 24, col 6.
   Executed then branch in "Dicho.java", line 27, col 26.

   [0.749 s 6233632 bytes]  failed
```

One can observe that ESC/Java2 is able to detect reasoning errors in specifications as well as in the code. No concrete counter-example is displayed, but the error is well underlined. One can also notice that the errors are displayed as warnings. ESC/Java2 suspects an error but does not guarantee it. Unfortunately, ESC/Java2 generally suspects too many errors: in most verification scenarii using ESC/Java2, ESC/Java2 displays many false warnings. The number of warnings can be restricted if the user spends time to understand the reasoning of the theorem prover.

Helping Simplify The way of formalising assertions may help the automatic verification: with the following formalisation (which is the one chosen is Figure 2.2), ESC/Java2 is not able to prove that the invariant holds.

```
@ requires a!=null &&
           (\forall int i ; 0 <= i & i < a.length-1 ;
                                 a[i] <= a[i+1]) ;
```

In fact, for instance, to prove $a[m] < x \Rightarrow (\forall k : 0 \leq k \leq m : a[k] < x)$, given that $(\forall i : 0 \leq i < a.\text{length} - 1 : a[i] \leq a[i+1])$, one must for each k, $0 \leq k < m$, show that $a[k] \leq a[k+1]$, $a[k+1] \leq a[k+2] ... a[m-1] \leq a[m]$, and given that $a[m] < x$, we conclude that $a[k] < x$.
This needs induction, what Simplify does not provide.

On the other hand, with the following formalisation, the theorem prover is able to check the verification condition.

```
@ requires a!=null &&
@          (\forall int i,j; 0 <= i & i < j & j < a.length ;
@                                a[i] <= a[j]);
```

Indeed, given $(\forall i : 0 \leq i < j < a.\text{length} : a[i] \leq a[j])$, Simplify just needs to recognise for each k, $0 \leq k < m$ that $a[k] \leq a[m]$ and given that $a[m] < x$, $a[k] < x$ by transitivity.

The ESC/Java2 verification does not reflect some principles of the structured programming method Without invariant, ESC/Java2 seems to be able to prove the algorithm correctness or to find errors. For example, if we initialise g to 1 instead of 0, the tool is able to find a problem in the iteration without any invariant:

```
Dicho.java:24: Warning: Array index possibly too large (IndexTooBig)
   if (a[m] < x) {g = m+1;
       ^
Execution trace information:
   Reached top of loop after 0 iterations in "Dicho.java",
   line 21, col 6.
```

Contrary to the invariant method, the initialisation and the iteration are not independent from each other; in the invariant method they are independent thanks to a complete invariant. In fact, ESC/Java is not designed to detect a too weak invariant. However, to let ESC/Java2 detect a too weak invariant, we can decompose a loop in three specified submethods corresponding to the Hoare propositions; for example, $\{Inv \text{ and not } H\}\ Iter\ \{Inv\}$ is implemented as follows:

```
  static int g;
  static int d;
  static boolean b;

/*@requires a != null  &&
             (\forall int i,j ; 0 <= i & i < j & j <= a.length ;
                                a[i] <= a[j]);
  @requires (0 <= g & g <= d & d <= a.length) ;
  @requires (\forall int i ; 0 <= i & i < g ;
                             a[i] < x) ;
  @requires (\forall int i ; d <= i & i < a.length ;
                             a[i] > x) ;
  @requires b ==> (\exists int j ; 0 <= j & j < a.length;
                                   a[j]==x) ;
  @requires ((b == false) & (g < d));

  @ensures (0 <= g & g <= d & d <= a.length) ;
  @ensures (\forall int i ; 0 <= i & i < g ;
                            a[i] < x) ;
  @ensures (\forall int i ; d <= i & i < a.length ;
                            a[i] > x) ;
  @ensures b ==> (\exists int j ; 0 <= j & j < a.length;
                                  a[j]==x);
  @*/
```

```
public  static boolean Iter(int[] a, int x)
{

     int m = (g + d) / 2 ;
     if (a[m] < x) g = m+1   ;
     else if (a[m] > x) d = m ;
     else b = true ;

}
```

where `g`, `d` and `b` are global declarations for the three methods. Using this way of writing each sequence of the loop, the verification guarantees[3] that the invariant is **complete**. However, we notice that this less natural way of writing a loop requires that the auxiliary variables used in the invariant are declared as global variables, and we need to write several times the invariant, in pre and/or postcondition of the three methods.

Badly defined assertions We can also notice that ESC/Java2 does not consider that an assertion may not be well defined; we mean that, for example, it does not check out-of-bound errors or division by zero in assertions. Simplify uses the first-order theory of arrays (with the two axioms "select and store") [35], where out-of-bounds are not considered as errors. Strict or non strict evaluations do not have the same behaviour as the evaluation of real Java expressions. For example, the assertion `x / 0 == 1 | b == b` is evaluated to true since `x / 0` is not illformed.

Several semantics are possible; as there is no formal, official specification of the JML language, we have to test particular cases to have an idea of the language semantics. It seems important for us to understand on which semantics the verification is based, and so, to have an idea of the translation of JML into the Simplify language.

3.1.3.2 Indian Exponentiation

Let us have a look to the ESC/Java2 behaviour when it analyses an algorithm involving multiplications.

```
/*@requires x != 0 && y >= 0 ;
  @ensures \result == Math.pow(x , y) ;
  @*/
  public /*@ pure @*/ static int algo(int x, int y)
  {
       int u = x ;
       int v = y ;
       int z = 1 ;
```

[3]assuming that ESC/Java is able to prove these assertions, which is not always true

```
/*@loop_invariant Math.pow(x , y) == z * Math.pow(u , v) ;
  @loop_invariant v >= 0 ;
  @*/
      while(v != 0){
         if (v % 2 == 1){
             z = z * u ;
             v = v - 1 ;
         }
         u = u * u ;
         v = v / 2 ;
      }
      return z ;
    }
```

A priori, this algorithm cannot be verified for two reasons:

The power operator is not a Java primitive The corresponding function in the `Math` class does not seem to be specified in JML and so it cannot be used by ESC/Java2. We can imagine defining the power relation, by means of a function head and axioms as follows:

```
//@ public static pure model int power(int x, int y);

/*@ axiom (\forall int x , y ;
    y == 0 ; power( x , y ) == 1) ;
  @ axiom (\forall int x , y ;
    y > 0 ;  power( x , y + 1 ) == x * power( x , y )) ;
  @*/
```

ESC/Java2 is not complete for non linear expression Unfortunately, given the characteristics of Simplify, for all exercises involving non linear multiplications, the tool gives spurious warnings.

However, we can define our own multiplication theory:

```
public static /*@ pure @*/ int mul(int x, int y)
    {return x*y;};

/* @ axiom (\forall int x ;  mul( x , 0 ) == 0) ;
   @ axiom (\forall int x , y ;
        mul( x + 1 , y ) == mul( x , y ) + y ;
   @ axiom (\forall int x , y ;
        mul( x , y ) == mul( y , x ));
   @ axiom (\forall int x , y , z ;
        mul( x , mul( y , z )) == mul( mul( x , y ) , z )) ;
   @*/
```

The last two axioms can be proved from the first two ones but this is not completely trivial. We update the multiplication operator `*` in the iteration and in the `power` axiom by calls of `mul` function, and we try to verify the algorithm using ESC/Java2: Simplify gets lost. On the other hand, if we delete the previous definitions and we give the following rules:

```
    /*@ axiom (\forall int x , y ;
        y == 0 ; power( x , y ) == 1) ;
      @ axiom (\forall int x , y;
        y % 2 != 1 ==>
        power( x , y ) == power( mul( x , x ) , y / 2 )) ;
      @ axiom (\forall int x , y , z ;
        y % 2 == 1 ==>
        mul( power( x , y ), z) ==
        mul( power( mul( x , x ) , y / 2) , mul( x , z ))) ;
      @*/
```

then ESC/Java2 is able to prove the algorithm in about 2.6 secondes if we unroll the loop 5 times. One can observe that these rules correspond to pattern-matching between equalities in the manual proof described in Section 2.1.2. These rules are elaborated by the user and may be unsound; ESC/Java2 may discharge false verification conditions by using these rules.

We also must notice, that ESC/Java2 does not detect any overflow, although the power of two integers may be larger than an integer: contrary to Chapter 2 where we consider that integers have an infinite domain, Java integers have a limited size.

3.1.3.3 The Next Permutation Problem

We now observe the behaviour of ESC/Java2 with the next permutation algorithm involving a complex specification and a decomposion into subproblems. The decomposition is explained in Section 2.1.5. We first verify the subproblem 1 consisting of finding the larger index i of the array such that $a[i] < a[i+1]$.

The Java code and the JML specifications follow. Again, the assertion expressing that the subarray $a[i+1..a.\text{length}-1]$ is decreasing, has been formatted to help Simplify verify the correctness of the algorithm. The same observation can be made for the second part of the invariant.

```
/*@requires (a != null) && (a.length > 0);
  @ensures  -1 <= \result & \result <= a.length - 2;
  @ensures  \result == -1 || a[\result] < a[\result + 1];
  @ensures (\forall int k ;
               0 <= k && k < a.length - 2 - \result ;
               a[k + 1 + \result] >= a[k + 2 + \result]) ;
  @*/

public /*@ pure @*/ static int sp1(int[] a) {
  int i = a.length - 2;
/*@loop_invariant -1 <= i & i <= a.length - 2;
  @loop_invariant (\forall int k ;
                       0 <= k && k < a.length - 2 - i ;
                       a[k + 1 + i] >= a[k + 2 + i]) ;
  @*/

  while (i != -1 && a[i] >= a[i + 1]) i--;

  return i;
}
```

Ranging the quantified variables from 0 may help the theorem-prover.

The invariant is not the key of verification In fact, the tool checks verification conditions but does not prove partial correctness using the Hoare method. Using the weakest precondition method [17] on a loop unrolled 3 times, the tool is able to prove the correctness of the subproblem for all array a such that a.length < 4; the tool does not guarantee correctness for an array a with a.length ≥ 4. For example, if the specifications are the following:

```
/*@requires (a != null) && (a.length > 4);
  @ensures  -1 <= \result & \result <= a.length - 2;
  @ensures  \result == 88 || a[\result] < a[\result+1];
  @ensures (\forall int k ;
  @            0 <= k && k < a.length - 2 - \result;
  @            a[k + 1 + \result] >= a[k + 2 + \result]);
  @*/
```

we need to unroll the loop at least 3.5 times to detect the error in postcondition.

```
SP1: sp1(int[]) ...
------------------------------------------------------------------------
SP1.java:19: Warning: Postcondition possibly not established (Post)
    }
    ^
Associated declaration is "SP1.java", line 4, col 10:
   @   ensures  \result == 88 || a[\result] < a[\result+1];
       ^
Execution trace information:
   Reached top of loop after 0 iterations in "SP1.java", line 14, col 8.
   Reached top of loop after 1 iteration in "SP1.java", line 14, col 8.
   Reached top of loop after 2 iterations in "SP1.java", line 14, col 8.
   Reached top of loop after 3 iterations in "SP1.java", line 14, col 8.
   Short circuited boolean operation in "SP1.java", line 14, col 23.
   Executed return in "SP1.java", line 18, col 8.
```

```
------------------------------------------------------------------------
    [0.56 s 6030952 bytes]  failed
```

It is said that the disjunction of the postcondition may not hold in the case where we need to iterate at least three times. So, when we iterate two times, ESC/Java2 does not detect any error. The verification is unsound; however, we admit that, in most of cases, if the parameter of unrolling is well chosen, this method gives a good feedback about the correctness of the algorithm. The same process is used to verify that the invariant holds.

Verification of the main algorithm A complete version of the main algorithm and its subproblems SP2 and SP3 with their JML specifications is given in Appendix A.1.2. ESC/Java2 is able to prove the subproblems (if specifications are formatted to help Simplify), but the main algorithm seems unprovable.

We present some non trivial steps in the elaboration of specifications of the main algorithm. As in Section 2.1.5, we need to formally express in precondition that `a` is a permutation of $0..a.\text{length}-1$ [4]. We propose the following JML precondition.

```
@requires (a != null) && (a.length > 0);
@requires (\forall int i ; 0 <= i && i < a.length ;
@           (\num_of int k ; 0 <= k & k < a.length ;
@                            a[k] == i)
@           == 1) ;
```

We express that for each value $0..a.\text{length}-1$, the number of occurrences in the array a is one. Unfortunately, this quantifier is not supported by ESC/Java2. In postcondition, we express that a is strictly greater than a_0 in the lexicographic order:

```
@ensures \result ==>
@             (\exists int i ; 0 <= i && i < a.length ;
@                  ((\forall int k ; 0 <= k && k < i ;
@                                  a[k] == \old(a[k]))
@               &&  a[i] > \old(a[i])));
```

Giving the code of the main algorithm, and each subproblem completely specified, ESC/Java2 is able to prove this assertion. We can also express that a is the smallest next permutation of a_0:

```
@ensures \result ==>
         (\forall int[] c;
          (c.length == a.length
           &&
```

[4]We consider a permutation of $0..a.\text{length}-1$ and not of $1..n$ for practical reasons: the first index of a Java array is 0

```
(\forall int i; 0 <= i && i < c.length;
        (\num_of int k ; 0 <= k & k < c.length;
                         c[k] == i)
        ==1)
 &&
(\exists int i; 0 <= i && i < a.length;
            ((\forall int k ; 0 <= k && k < i;
                              c[k] == \old(a[k]))
              &&
              c[i] > \old(a[i]))))
 ==>
(\exists int i; 0 <= i && i < a.length;
            ((\forall int k ; 0 <= k && k < i  ;
                                 c[k] == a[k])
          &&
            c[i] >= a[i])));
```

Again, one can observe that JML is very expressive, we can quantify on objects such as arrays. But ESC/Java2 is not able to prove this assertion.

Another way for specifying the main algorithm, is to translate the algorithm in postcondition (in a declarative way) : we may express that the array a is divided in two parts, the first, $a[0..i-1]$ is unchanged, the second, $a[i..a.\text{length}-1]$ is decreasing, with $a[i] > a_0[i]$ and $a[i]$, the smallest greater than $a_0[i]$:

```
@ensures \result ==>
 (\exists int i; 0 <= i & i <= a.length-1;
               \old(a[i]) < a[i]
           &&
               (\forall int j; 0 <= j & j < i;
                                \old(a[j]) == a[j])
           &&
               (\forall int j; i+1 <= j & j < a.length-1 ;
                                a[j]) < a[j+1])
           &&
               (\forall int j; i+1 <= j & j <= a.length-1;
                        a[j] > \old(a[i]) ==> a[j] > a[i])
 );
```

This part of postcondition is the translation of the next permutation algorithm. But who does guarantee that this assertion expresses correctly the next permutation? It does not guarantee that the algorithm actually computes the next permutation if we verify it according to these specifications. This kind of verification is just a way to be more confident in the algorithm but it does not prove that the algorithm really computes the next permutation. Anyway, ESC/Java2 is not able to prove it; the manual proof in Section 2.1.5 suggests that it would be difficult for an automatic theorem prover to prove that the subarray $a[i+1..a.\text{length}-1]$ is decreasing.

3.1.4 Conclusion

JML allows us to write convenient specifications, and keywords exist to specify preconditions, postconditions and loop invariants. There is even a keyword to provide a variant. Nevertheless, it is not completely adequate to learn a strict structured programming methodology for a number of reasons.

- First, the use of the invariant method is not encouraged: ESC/Java2 does not need an invariant to prove correctness with respect to specifications. Moreover, ESC/Java2 does not help us to write "good" (i.e., complete) invariants since it does not check Hoare's propositions independently.
- The feedback is not always very informative. Too often ESC/Java2 gives false warnings, and sometimes it forgets errors. Moreover, ESC/Java2 does not display verification conditions in a readable way.
- ESC/Java2 does not provide concrete counter-examples.
- Finally, the "pragmatism" of the ESC/Java2 design has a number of unpleasant consequences: since it is not possible to exactly know a priori what kind of assertions can be proven or falsified, users may spend too much time using the tool by trial and errors, which is exactly the opposite of our teaching goals.

A new European project [45] is improving ESC/Java2: instead of working with a single theorem prover, it proposes to couple multiple provers (even concurrently) and the choice of prover(s) is based upon context. New logics encoded in PVS and Coq will increase ESC/Java2 soundness, completeness, and performance. With these improvements, ESC/Java2 should be reevaluated with respect to the pedagogical goals.

In the next Section, we present another tool which follows a different philosophy: verification must be very precise; it cannot be unsound and must lead to complete verifications, the price to pay is that the tool is partially interactive.

3.2 Spark

According to John Barnes's book [3], Spark is a high level programming language designed to write software for applications that cannot fail. It is used in safety critical applications where life and limb are at risk if the program is in error, or in security applications where access and integrity of information is critical. When the user is an expert in theorem proving, Spark tools enable to prove partial correction of programs. Besides, Spark encourages the development of programs in an orderly manner with the aim that the program should be correct by construction. In application areas as avionics and railway signaling [1], it is argued that, when SPARK is used, not only the program is more likely to be correct, but the overall cost of development is actually less in total after all testing and integration phases are taken into account.

SPARK is also used in the academical context [36]; indeed, SPARK offers a small language that can be used to teach the principles of design-by-contract (which is a fashionable word for structured programming), static analysis, formal verification and safety- and security-critical software development [48]. The book [3] is quite pedagogical and explains the SPARK language and technology.

As the SPARK philosophy corresponds exactly to our objectives in the structured programming method learning, we propose to analyse to what extend the SPARK tools are adapted to help the teacher and the student in the concerned methodology. To evaluate the adequacy of this tool in our learning context, we would like to approximate the set of algorithms that can be verified fully automatically, and we would like to appreciate the required level of expertise of the user, when algorithms cannot be verified automatically.

3.2.1 Overview

3.2.1.1 The Programming Language

The SPARK language comprises a kernel which is a subset of Ada [25]; It provides additional features inserted as annotations in the form of Ada comments. The annotations are divided into two categories: the first one is concerned flow analysis, and visibility control; the second one is concerned with formal proof. Notice that, although the kernel language omits many features of full Ada, it stays nevertheless a rich language containing for example a full capability for defining Abstract Data Types that we do not investigate. To remain on our research track, we focus on the part of the language related to the structured programming method explained in Chapter 2.

The first category of annotations (named *kernel annotation*s) can express the use of global variables by subprograms, to specify the information flow between their imports and exports via both parameters and global variables. For example, the following method declaration, together with its kernel annotations,

```
procedure Algo(X , Y : in My_Integer ;
                     Z : out Integer) ;
    --# derives Z from X , Y ;
```

expresses that the formal parameters X and Y are constants initialized by the value of the associated actual parameter, and Z is an uninitialized variable, that can be updated.
The annotation "derives Z from X,Y" expresses that the value of Z will be obtained from the values of X,Y.

Pre and post conditions are in the second category (named *proof annotations*); unlike the kernel annotations, proof annotations are optional and are written in an expression language which is an extension of Ada. We can refer to the initial values of variables (X~ denotes the inital value of X); we can use logical implication, universal and existential quantifiers. The following proof annotation expresses that the subarray A[I+2..N] stays unchanged:

```
 --# (for all K in Integer range I+2..N => (A(K) = A~(K)));
```

K in Integer is the declaration of the quantified variable, the range is optional and the body of the quantified expression compares the values at each array index, which satisfies the range, with their respective initial values. We can also declare proof functions for use in annotations as mathematical function which we define by means of a collection of proof rules. For example, we declare a function Perm to express that an array A is a permutation of an array B:

```
 --# function Perm(A, B : Array_Type) return Boolean;
```

and so we can write, for example, in a proof annotation that A is a permutation of the initial array A~ : --# Perm(A , A~).
Precondition is determinated by the keyword # pre, postcondition, with the keyword # post. Within subprogram bodies, additional proof annotations are used for

- code cutpoints (e.g. loop invariant) via # assert annotations;
- well-formation checks, via # check annotations

Each of such assertions specifies a relation between program variables which must hold at that precise point, whenever execution reaches it. Assert annotations generate a cut point on the flow chart of the subprogram, whereas check annotations do not.

```
package Exponentiation
is
    subtype My_Integer is Integer range 0..5 ;
    procedure Algo(X , Y : in My_Integer ;
                     Z : out Integer) ;
    --# derives Z from X , Y ;
    --# pre Y >= 0 ;
    --# post Z = X ** Y ;
end Exponentiation;

package body Exponentiation
is
   procedure Algo(X,Y: in My_Integer; Z: out Integer)
   is
    U : Integer;
    V : My_Integer;
    begin
      U := X ;
      V := Y ;
      Z := 1 ;
      loop
        --# assert X ** Y = Z * U ** V and V >= 0 ;
      exit when V = 0 ;
       if V mod 2 = 1 then Z := Z * U ; V := V - 1 ; end if ;
       U := U * U ;
       V := V / 2 ;
      end loop;
   end Algo;
end Exponentiation;
```

Figure 3.1: The Indian exponentiation algorithm

To present the SPARK tools, we refer to the example of the Indian exponentiation, a SPARK version of which is depicted in Figure 3.1; this consists in two packages; the first one defines the method specification, the second one contains the method implementation. This SPARK version of specification and implementation is very similar to that elaborated in Section 2.1.2 in Figure 2.1. Apart from the syntax, the only difference lies in the fact that, in the SPARK version, we define a subtype `My_Integer` that is not defined in Figure 2.1. The algorithms that we mention in Chapter 2 do not consider overflow for integers (they are treated as mathematical integers); actually any programming language has to consider that the set of integers has a limited size. In this algorithm, overflow can be source of error: if we compute the exponentiation of two integers, the result may be larger that an integer and even larger than a long. To simplify the problem, we take the option of limiting the size of the data.

3.2.1.2 The Examiner

Flow analysis The Examiner is the main Spark tool, it checks conformance of the code to the rules of the kernel language and it checks consistency between the code and the embedded annotations by control, data and information flow analysis. So, the Spark language with its *kernel annotations* ensures that a program cannot have certain errors related to the flow of information. The Examiner can detect the use of uninitialized variables and the overwriting of values before they are used. The techniques used are based on [34]. In order to do so, Spark accepts only assignments, sequential composition, conditional statements and loops (to be precise, Spark also accepts multi-exit loops and case statements, which are statements that can be composed out of the primitive compound statements by various transformations). The information flow analysis is based on a number of relationships between variables and expressions. The expressions depend of the variables and the variables depend on the expressions. As a consequence, the final values of the variables depend upon the initial values of the variables through the intermediary of the expressions. To develop formulae for the two relationships between expressions and variables, they define, for each statement, a binary matrix. Then, they can easily compute the overall relationship, using operations on the matrices: the reasoning is based on abstract interpretation.

Generating Verification Conditions In case no problem occurs in the flow analysis, given proof annotations, the Examiner generates, in a specific file, all the verification conditions using the weakest precondition method. For each Hoare proposition, the Examiner generates formulas of the form H_1 & H_2 & $\ldots \rightarrow C_1$ & $\ldots$. The verification conditions produced by the Examiner are expressed in a form that can be manipulated by the automatic tools that we describe later. The employed notation is an extended form of the language FDL (Functional Description Language). The translation details of Spark expressions into FDL expressions is provided in the book [3]. The verification condition below is the last one (the 16th), this corresponds to the Hoare proposition $\{Inv$ and $H\}$ *clot* $\{$ *Post* $\}$;

```
procedure_algo_16.
H1:    x ** y = z * u ** v .
H2:    v >= 0 .
H3:    x >= my_integer__first .
H4:    x <= my_integer__last .
H5:    y >= my_integer__first .
H6:    y <= my_integer__last .
H7:    y >= 0 .
H8:    v >= my_integer__first .
H9:    v <= my_integer__last .
H10:   v = 0 .
```

```
        ->
C1:    z = x ** y .
```

If we prove that, assuming the clauses `H1... H10`, the conclusion `C1` holds, we guarantee that the proposition { *Inv* and *H*} *Clot* { *Post* } holds. Figure 3.2 shows the verification condition corresponding to the iteration when *v* is odd. One can observe that verification conditions may easily become very large.

Verification conditions are also generated for runtime checks: index range, division checks and overflow checks. In the example, we have for each assignment of `U`, `V` and `Z` a verification condition to check whether there is no runtime error. To achieve this, the Examiner automatically inserts a proof annotation just before each assignment; and then, generates verification conditions using the weakest precondition method. It is the reason why in this example, there are 16 verification conditions.

```
...
--# check U * U in Integer;
U := U * U;
--# check V div 2 in My_Integer and 2 <> 0;
V := V div 2;
..
```

Here is the verification condition checking whether there is no runtime error with the assignment `u := u * u`.

```
H1:    x ** y = z * u ** v .
H2:    v >= 0 .
H3:    x >= my_integer__first .
H4:    x <= my_integer__last .
H5:    y >= my_integer__first .
H6:    y <= my_integer__last .
H7:    y >= 0 .
H8:    v >= my_integer__first .
H9:    v <= my_integer__last .
H10:   not (v = 0) .
H11:   v >= my_integer__first .
H12:   v <= my_integer__last .
H13:   2 <> 0 .
H14:   v mod 2 = 1 .
H15:   z >= integer__first .
H16:   z <= integer__last .
H17:   u >= integer__first .
H18:   u <= integer__last .
H19:   z * u >= integer__first .
H20:   z * u <= integer__last .
```

```
procedure_algo_5.
H1:    x ** y = z * u ** v .
H2:    v >= 0 .
H3:    x >= my_integer__first .
H4:    x <= my_integer__last .
H5:    y >= my_integer__first .
H6:    y <= my_integer__last .
H7:    y >= 0 .
H8:    v >= my_integer__first .
H9:    v <= my_integer__last .
H10:   not (v = 0) .
H11:   v >= my_integer__first .
H12:   v <= my_integer__last .
H13:   2 <> 0 .
H14:   v mod 2 = 1 .
H15:   z >= integer__first .
H16:   z <= integer__last .
H17:   u >= integer__first .
H18:   u <= integer__last .
H19:   z * u >= integer__first .
H20:   z * u <= integer__last .
H21:   v >= my_integer__first .
H22:   v <= my_integer__last .
H23:   v - 1 >= my_integer__first .
H24:   v - 1 <= my_integer__last .
H25:   v - 1 >= my_integer__first .
H26:   v - 1 <= my_integer__last .
H27:   v - 1 > 0 .
H28:   u >= integer__first .
H29:   u <= integer__last .
H30:   u * u >= integer__first .
H31:   u * u <= integer__last .
H32:   v - 1 >= my_integer__first .
H33:   v - 1 <= my_integer__last .
H34:   (v - 1) div 2 >= my_integer__first .
H35:   (v - 1) div 2 <= my_integer__last .
H36:   2 <> 0 .
        ->
C1:    x ** y = z * u * (u * u) ** ((v - 1) div 2) .
C2:    (v - 1) div 2 >= 0 .
C3:    x >= my_integer__first .
C4:    x <= my_integer__last .
C5:    y >= my_integer__first .
C6:    y <= my_integer__last .
C7:    y >= 0 .
```

Figure 3.2: The verification condition corresponding to the iteration when v is odd

```
H21:   v >= my_integer__first .
H22:   v <= my_integer__last .
H23:   v - 1 >= my_integer__first .
H24:   v - 1 <= my_integer__last .
H25:   v - 1 >= my_integer__first .
H26:   v - 1 <= my_integer__last .
H27:   v - 1 > 0 .
H28:   u >= integer__first .
H29:   u <= integer__last .
        ->
C1:    u * u >= integer__first .
C2:    u * u <= integer__last .
```

The reader who proves this verification condition can observe that it is important to reduce the domains of the data `x` and `y`.

3.2.1.3 The Simplifier

The SPADE Automatic Simplifier (usually referred to as just the Simplifier) is an automatic tool that tries to reduce a verification condition to true. The fundamental algorithm employed by the Simplifier consists, for each verification condition, in a sequence of the processes described below; some of them are repeated several times. Notice that the algorithm always terminates but it does not guarantee to find a proof.

- **Scalar constant substitution:**

  Replacing the constants by their value.

  In the example, `integer__first` and `integer__last` are replaced resp. by `-2**31` and `2**31-1`;

  `My_integer__first` and `My_integer__last` are resp. replaced by 0 and 5.

- **Expression simplification:**

  Trying to evaluate expressions by using a list of simplification rules;

  for examples, `not true` is evaluated to `false`,

  `X + 0` is evaluated to `X`,

  `2 + 3` is evaluated to `5`,

  `X ** 0` is evaluated to 1.

  The Simplifier goal is to try to evaluate the conclusions of a verification condition to `true`.

- **Global logical simplification:**

  Moving negation inwards as far as possible, splitting up conjunctions and adding new consequents.

- **Proof conclusion:**

  The inference engine of Simplifier tries to deduce conclusions from the existing hypotheses.

- **Construction of standard forms:**

  Multiplying out terms such as `(a + b) * (d + c)`, collecting like terms and ordering sums of products lexicographically

- **Search for a contradiction:**

  If the inference engine cannot eliminate all the conclusions, the next strategy is to search for a contradiction in the hypothesis of the verification condition; if it finds a contradiction, the verification condition holds since `false` → `X` is true.

  To perform it, the tool looks for mutually exclusive hypotheses (for each hypothesis `P`, it attempts to infer `not P`); it tries to find a variable on an empty range; it constructs a collection of joins (a join is formed by adding or subtracting two relational expressions in a sound way).

- **Expression reduction:**

  Eliminating hypotheses `true`, repeated hypotheses, redundant hypotheses.

  Reducing $A \rightarrow B$ to B if the engine has inferred A to `true`.

- **Proof framing:**

  For a conclusion with one specific pattern, like `for_all(x:t,p(x))` or `p -> q`, the Simplifier has heuristics for breaking the conclusion into subgoals.

- **Application of user-defined rules:**

  If, after applying all of the preceding phases, there are still unproven conclusions in the current verification condition, the Simplifier attempts to make use of the user-defined rules to prove each of the remaining conclusions in turn; it is its final attempt to complete the proof of the current verification condition. We give some examples of user-defined rules later in this section.

Let us submit the verification condition `algo_16` to the Spade Automatic Simplifier for reducing it to simply

```
***true.  /*all conclusions proved*/
```

For the particular case, it proceeds like this (it is described in the log file):

1. The Simplifier applies substitution rules:

   it instantiates `my_integer'first` and `my_integer'last`,

2. Then, it eliminates all redundant hypotheses,
3. Then, it replaces all occurrences of `v` by `0`,

   the hypothesis `H10` is eliminated.

   In `H1`, `u ** 0` is simplified to `1` and `z * 1` is simplified to `z`.

   The new `H1` becomes `x ** y = z`.
4. Then, it replaces all occurrences of `z` by `x ** y`.

   `H1` is eliminated.

   We get `C1:  x ** y = x ** y.`
5. Finally `C1` is simplified to `true` and the verification condition is proved.

Of course, it is common that the Simplifier is unable to reduce a verification condition to true. If it does not succeed, it provides the initial verification condition reduced to an intuitive simpler form, which is not necessarily the last form obtained by the algorithm.

The verification condition `algo_5` in Figure 3.2 cannot be reduced to true: in the simplified verification condition `algo_5`, one can observe that one of the 7 conclusions has not been proved. For better readability, some hypotheses concerning the variables domains have not been displayed.

```
procedure_algo_5.
H1:    x ** y = z * u ** v .
H7:    y >= 0 .
H14:   v mod 2 = 1 .
H27:   v - 1 > 0 .
       ->
C1:    x ** y = z * u * (u * u) ** ((v - 1) div 2) .
```

To help the Simplifer, the user can provide *user-defined rules*: *inference rules* and *substitution rules*. For this example, we provide the following additional rule:

```
myexpo(4): Z * X ** Y may_be_replaced_by
                     Z * X * (X * X) ** ((Y - 1) div 2)
           if [Y mod 2 = 1] .
```

And the Simplifier is able to prove the conclusion `C1`. On the other hand, if we give the rule `myexpo(4)'`, the Simplifier is not able to prove `C1`.

```
myexpo(4)':  X ** Y may_be_replaced_by
                      X * (X * X) ** ((Y - 1) div 2)
          if [Y mod 2 = 1] .
```

Besides, it brings risk that the user may add rules that are not sound, and use these unsound rules to discharge verification conditions that are not provable. Ideally, the user should construct a proof with the Proof Checker, with other tool or, at least, manually.

3.2.1.4 The Proof Checker

The Proof Checker is an interactive program; this enables the user to direct the Checker to explore the use of various strategies and rules on the condition to be proved. It is designed around first-order predicate calculus with classical logic. It does not permit quantification over objects such as arrays, records, etc. Importantly, we must be careful about what we are claiming to have proved: we have to assume that all subexpressions occurring in a verification condition are well defined. Thus, if we have a hypothesis `c div d = x`, there is an assumption that `c div d` is defined (i.e, `d` is not zero); but imagine we also have the hypothesis that `d = 0`, then classical logic ceases being applicable, because we have a hypothesis whose value is undefined.

The Proof Checker proceeds by applying various strategies and rules. The rules are the hard facts available to the Checker whereas strategies are dynamical approaches available. The user directs the proof process by telling the Checker which strategies to apply. Available strategies include

- simple inference by pattern matching against the rules,
- deduction using truth tables,
- proof by cases: cases for `x = 0` and `x >= 0` might need different approaches,
- proof by contradiction,
- proof by induction.

To give an idea of the way the user interacts with the Checker, we prove the user-defined rule `myexpo(4)`; this example mainly applies substitution rules, more interesting proof strategies are shown later in the section.

```
H1:    y mod 2 = 1 .
       ->
C1:    z * x ** y  = z * x * (x * x) ** ((y - 1) div 2) .
```

To prove it, we need the following rules wich are in the database, the rules are grouped in families; they are identified by the name of their family and a number.

```
exp(8):     X ** 1       may_be_replaced_by   X .
exp(9):     X ** 2       may_be_replaced_by   X * X .
commut(2):  A * B        may_be_replaced_by   B * A .
assoc(1):   (A + B) + C  may_be_replaced_by   A + (B + C)
minus(1):   X - X        may_be_replaced_by   0 .
minus(7):   A + (-B) & A - B are_interchangeable .
intdiv(11): A * B div B  may_be_replaced_by   A  if [B <> 0] .
```

The keyword `may_be_replaced_by` implies one-way replacement only and `are_interchangeable` implies remplacement in both ways. Notice that `commut(2)` is a one-way rule covering both directions, `intdiv(1)` has a condition. But we also need rules that do not exist in the set of rules of the Checker:

```
myexpo(1):   X ** (Y * Z)  may_be_replaced_by  (X ** Y) ** Z .
myexpo(2):   X ** (Y + Z)  may_be_replaced_by  X ** Y * X ** Z .
myexpo(3):   X mod 2 = 1   may_be_replaced_by  X = K * 2 + 1 .
```

The database evolves over the years from Spark experience with projects; it is the reason why, for some operators, the set of rules contains many properties, and many rules quite easy to derive from the others, whereas for other operators, such as the exponentiation, there exist only few rules.

In fact, we can consider that the proof of `myexpo(4)` is made by decomposition into subgoals which are `myexpo(1)`, `myexpo(2)`, `myexpo(3)`. So, to be sound and complete, we need to prove all these three assumptions before proving `myexpo(4)`. As an example of proof by induction, the proof of `myexpo(2)` is given in Appendix A.2. Let us develop the proof of `myexpo(4)`. The interaction with the Checker for the first two steps is depicted in the real form. The next steps are described in an adapted, more readable way.

```
CHECK|:replace H#1
OLD EXPRESSION : y mod 2 = 1
Pattern?  y mod 2 = 1.
Subexpression is y mod 2 = 1
Change this expression (yes/no)? yes
Type new subexpression pattern t * 2 + 1 = y .
By which rule? myexpo.
myexpo(3) allows y mod 2 = 1 to be replaced by t * 2 + 1 = y directly
The only possible replacement for y mod 2 = 1 is:
    t * 2 + 1 = y
           according to rule myexpo(3)
Proceed(yes/no)? yes
NEW EXPRESSION : t * 2 + 1 = y
Is it OK(yes/no)? y
Replace more (yes/no)? no
OK

CHECK|: replace c#1.
```

```
OLD EXPRESSION : z * x ** y  = z * x * (x * x) ** ((y - 1) div 2) .
Pattern? y
Subexpression is y
Change this expression (yes/no)? yes
Change which occurrence (number/none/all)? 1.
...
```

At this moment, we replace the first occurrence `y` in `C1` by `t * 2 + 1` by doing an equality substitution. New `C1` is

`z * u ** (t * 2 + 1) = z * x * (x * x) ** ((y - 1) div 2)` .

According to `myexpo(2)`, we replace the pattern

`x ** (X + Y)` by `x ** X * x ** Y`.

By the predefined rule `exp(8)`, `x ** 1` is simplified to `x`.
We replace the pattern `t * 2` by `2 * t` using `commut(2)`.
We replace the pattern `x ** (2 * t)` by `(x ** 2) ** t` using `myexpo(1)`.
We replace the pattern `x ** 2` by `x ** x` using `exp(9)`.
The current `C1` is

`z * (x * x) ** t * x = z * x * (x * x) ** ((y - 1) div 2)`

After two commutations, `C1` becomes

`z * x * (x * x) ** t = z * x * (x * x) ** ((y - 1) div 2)`

Then, we prove `t * 2 = y - 1` using `assoc(1), minus(1), minus(7)`.
We replace the pattern `y - 1` in `C1` by `t * 2`, by doing an equality substitution.
Finally, `t * 2 div 2` can be replaced by `t` using `intdiv(11)`.
We get `C1:  z * x * (x * x ) ** y = z * x * (x *x) ** t` which is simplified to `true`.

One can observe that encoding this proof in the Checker is tedious, each step, as small as it can be, has to be formally justified. The command to let the Checker working by itself is `done`, but the Checker's inference engine is not as clever as we might expect.

3.2.2 Experimentation on some Examples

In this section, we evaluate the adequacy of these tools to support the methodolology of learning the structured programming method. Through two examples that we have studied in Chapter 2, we analyse if all the aspects of the methodology can be easily used with Spark. We first analyse the insertion sort algorithm, then we tackle the next permutation algorithm whose manual correctness proof was far from simple.

3.2.2.1 The Insertion Sort Algorithm

Decomposition into subproblems and the assertions expressivity In the abstract package (see Figure 3.3), we define the decomposition into two subproblems. For the main problem `Sort`, one can observe that we can easily express in postcondition that the result array `A` is sorted by using a

```
package triinsert is
   N : constant := 10;
   subtype Index_Type is Integer range 1..N;
   type Array_Type is array(Index_Type) of Integer;

   --# function Perm(A, B : Array_Type) return Boolean;

   procedure SP(A : in out Array_Type; I: in  Integer) ;
   --# derives A from A, I;
   --# pre 0 <= I and I <= N-1 and
   --#    (for all K in Index_Type range 1 .. I-1
   --#               => (A(K) <= A(K+1)));
   --# post (for all K in Index_Type range 1 .. I
   --#                 => (A(K) <= A(K+1)))
   --#      and Perm(A , A~) and
   --#      (for all K in Index_Type range I+2 .. N
   --#                 => (A(K) = A~(K)));

   procedure Sort(A : in out Array_Type);
   --# derives A from A;
   --# post  (for all K in Index_Type range 1 .. N-1
   --#                  => (A(K) <= A(K+1)))
   --#       and Perm( A , A~);
end triinsert;
```

Figure 3.3: Insertion sort specifications

```
package body triinsert is
  procedure Sort(A : in out Array_Type) is
      I : Integer;

   begin -- Sort --
      I:= 0;
      loop
         --# assert 0 <= I and I <= N and
         --#   (for all K in Index_Type range 1 .. I-1
         --#              => (A(K) <= A(K+1)))
         --#   and Perm(A , A~);
         exit when I =  N ;
         SP(A,I);
         I := I+1;
      end loop;
   end Sort;
end triinsert;
```

Figure 3.4: Insertion sort Algorithm

universal quantifier; besides SPARK allows us to formally express that the result array `A` is a permutation of the initial array `A~` thanks to a proof function. The specification of the subproblem `SP`, whose goal is to insert the element `A(I)` at the right place in the subarray `A[1..I]`, is also expressed in a similar way as in Section 2.1.4. In the abstract package, we also define the variables types, the array size and we declare the proof function `Perm`. The implementation of the main problem stands in the package body (see Figure 3.4) and the loop invariant is inserted in the code.

Proof of the correctness using the invariant method SPARK follows the principles of the stuctured programming method: independently, for both subproblems, the Examiner generates (in two separate files) verification conditions using the weakest precondition and the invariant. Let us observe the verification condition of the loop transformation of the main algorithm. For better readability, some hypotheses concerning the variables domains have not been displayed and redundant hypotheses have been omitted as well.

```
procedure_sort_3.
H1:    0 <= i .
H3:    for_all(k_: integer, ((k_ >= 1) and (k_ <= i - 1))
       -> (element(a, [k_]) <= element(a, [k_ + 1]))) .
H4:    perm(a, a~) .
H14:   i <= n - 1 .
H16:   for_all(k_: integer, ((k_ >= 1) and (k_ <= i))
       -> (element(a__1, [k_]) <= element(a__1, [k_ + 1]))) .
H17:   perm(a__1, a) .
```

```
H18:   for_all(k_: integer, ((k_ >= i + 2) and (k_ <= n))
       -> (element(a__1, [k_]) = element(a, [k_]))) .
        ->
C1:    0 <= i + 1 .
C2:    i + 1 <= n .
C3:    for_all(k_: integer, ((k_ >= 1) and (k_ <= i + 1 - 1))
       -> (element(a__1, [k_]) <= element(a__1, [k_ + 1]))) .
C4:    perm(a__1, a~) .
```

The verification conditions that we prove in Section 2.1.4 are not exactly the same as those generated by SPARK because, in order to define them, we use the symbolic execution method and SPARK uses the weakest-precondition method. By experiment, these verification conditions are very similar, there is just less renaming with the *wp* method.

To be verified, the predicate `Perm` must be defined: it is declared in the abstract package in Figure 3.3, but it also needs to be defined as an equivalence or one can axiomatize its definition by a set of rules. The second method is often more rewarding since the rules can be efficiently used by the Simplifier and the Proof Checker while a definition as an equivalence can be operationally useless. However, incorrect rules may allow the user to “prove the correctness” of a wrong program. Hence, a rigorous approach would be to formally prove that rules are valid according to a definition as an equivalence. But, since such a proof can be difficult or tedious, it is often omitted in practice. We now display a set of rules for the predicate `perm(X,Y)`.

```
perm(1): perm(update(update(A,[I],X),[J],Y),B)
             may_be_deduced_from
                 [perm(update(update(A,[I],Y),[J],X),B)].
perm(2): perm(A,A) may_be_deduced .
perm(3): perm(A,B) may_be_replaced_by perm(B,A).
perm(4): perm(A,B) may_be_deduced_from [perm(A,C),perm(C,B)].
```

The first rule expresses that when two elements are exchanged, the array remains a permutation. The second rule stands for the identity of `perm`; the third rule expresses that `perm` is symmetric. `perm(4)` expresses the transitivity of the relation.

Using these rules, the Simplifier is able to verify all the conclusions of this verification condition. And also, most of the verification conditions of the main problem are automatically verified. Amazingly, the Simplify is not able to simplify the following condition to true:

```
C1:    for_all(k_ : integer, 1 <= k_ and k_ <= - 1 ->
               element(a, [k_]) <= element(a, [k_ + 1])) .
```

We do not detail here the interactive proof of this condition because we let the next example explore the possibilities of the Proof Checker.

Anyway, the feedback of SPARK is not clear, it does not give any warning, it just says when the Automatic simplifier has finished. Some verification conditions are completely simplified and for other verification conditions, the simplification has not succeeded. In the last case, it is not clear if it is because the verification condition is false, or because the tool needs additional rules, or even because the tool is not clever enough. The user has to analyse it manually.

3.2.2.2 The Next Permutation Problem

Decomposition into subproblems and assertions expressivity
Again, we have an abstract package (in Figure 3.5) which describes the several subproblems with there specificiations such as they are described in Section 2.1.5. The main problem can be formalised: to constraint the array `A` to be a permutation of `1..N`, we define a proof function `perm1N(A)`; and in postcondition, to express that `A` is the next permutation of `A~`, we use a trick: a formal parameter `C` in the procedure `Main` which is just mentioned in the specification, to say that for all array `C` the written postcondition is satisfied. `greater`, and `greaterEq` are two proof functions expressing lexicographic relations between arrays of integers. Notice that the Examiner, analysing the flow, gives some warnings because `C` is not used in the code.

Proof of the correctness This example highlights the Simplifier limits because, even for simple subproblems such as the subproblems `SP1` and `SP2`, the Simplify cannot manage quantified expressions. Each of the subproblems must be proved with the interactive Proof Checker. To show an example of using the interactive Proof Checker with verification conditions involving quantifiers, we propose to prove the first subproblem, its goal is to find the smallest index `i` of an array `a` such that `a[i] < a[i+1]`.

The proof of the correctness of the subproblem SP1 To guide the proof checker, we need to elaborate a good manual proof first. Thus, given the algorithm with its specification in Figure 3.6, we provide a manual proof of its correctness. Afterwards, we show how it could be formalised with SPARK.

The manual proof To demonstrate the partial correctness of the SP1 algorithm, let us prove the three Hoare propositions by symbolic execution.

1. $\{a \text{ initialised}\}$ `i := n - 1` $\{Inv\}$

   The execution of the initialisation terminates with $i = n - 1$;

   with a initialised and $i = n - 1$, the invariant holds:

   - $0 \leq i \leq n - 1$ and

```
package permutsuivante is
   N : constant := Integer(100) ;
   subtype Index is Integer range 1..N ;
   type Array_Type is array(Index) of Index ;

   --# function perm1N(A : Array_Type) return Boolean ;
   --# function greater(A1 : Array_Type ; A2: Array_Type)
   --#          return Boolean ;
   --# function greaterEq(A1: Array_Type ; A2: Array_Type)
   --#          return Boolean ;

   procedure SP1(A : in Array_Type; I : out Integer) ;
   --# derives I from A ;
   --# post 0 <=I and I < N
   --#      and (for all K in Index range  I+1.. N-1
   --#                      => A(K) >= A(K+1))
   --#      and (I = 0 or A(I) < A(I + 1)) ;

   procedure SP2(A : in Array_Type ; I : in Integer
                                   ; J : out Integer) ;
   --# derives J from I , A ;
   --# pre  0 < I and I < N
   --#      and (for all K in Index range  I+1.. N-1
   --#                     => A(K) >= A(K+1))
   --#      and A(I) < A(I+1) ;
   --# post I < J and J <= N and A(J) > A(I)
   --#      and (J = N or A(J + 1) <= A(I)) ;

  procedure SP3(A : in out Array_Type; I : in Integer) ;
  --# derives A from A,I ;
  --# pre  1<= I and I< N ;
  --# post (for all K in Index range  I + 1..N
  --#                 => (A(K) = A~(N - (K - (I + 1)))))
  --#      and (for all K in Index range  1..I
  --#                     => (A(K) = A~(K))) ;

  procedure Main(A : in out Array_Type ;
                 C : in out Array_Type ; B : out Boolean) ;
  --# derives A , B from A &
  --#         C from C ;
  --# pre  perm1N(A) ;
  --# post (B -> (perm1N(A) and greater(A , A~)
  --#            and  ( (perm1N(C) and greater(C , A~))
  --#                    -> greaterEq(C , A))
  --#            ))
  --#       and
  --#      (B = false -> (for all K in Index_Type range 1..N-1
  --#                                => (A(k) > A~(K+1)))) ;

end permutsuivante;
```

Figure 3.5: The next permutation specification

- Declarations:
  var a : array[1..n] of integer ; (input)
  var i : integer ; (output)
- Pre: a initialised
  Post: a unchanged and $0 \leq i \leq n-1$
  and $(\forall k : i+1 \leq k \leq n-1 : a[k] \geq a[k+1])$
  and $(i = 0$ or $a[i] < a[i+1])$
  Invariant: a unchanged and $0 \leq i \leq n-1$
  and $(\forall k : i+1 \leq k \leq n-1 : a[k] \geq a[k+1])$
- Init: `i := n - 1`
  Iter: `i := i - 1`
  H: `i = 0 || a[i] < a[i+1]`

Figure 3.6: The algorithm to find the smallest index `i` such `a[i] < a[i+1]`

  - $(\forall k : n \leq k < n : a[k] \geq a[k+1])$ is true because there is no k such that $n \leq k < n$.

2. $\{Inv$ and not $H\}$ `i := i - 1` $\{Inv\}$
   Let i_1 be the initial value of i; by hypothesis,
   (1) $0 < i_1 \leq n-1$ and
   (2) $(\forall k : i_1+1 \leq k < n : a[k] \geq a[k+1])$ and
   (3) $a[i_1] \geq a[i_1+1]$
   The execution of the iteration terminates with $i = i_1 - 1$ and the invariant holds:
   - $0 \leq i \leq n-1$, because of **(1)**:
     $0 \leq i_1 - 1$, i.e., $0 < i_1$
     and $i_1 - 1 \leq n-2$, which implies $i_1 - 1 \leq n-1$
   - $(\forall k : i+1 \leq k < n : a[k] \geq a[k+1])$ holds, i.e:
     $(\forall k : i_1 \leq k < n : a[k] \geq a[k+1])$ holds:
     Indeed, this last quantified assertion is the same as **(2)** except that the range of the quantification has an extra value i_1; but the case of the extra value i_1 is covered by the relation **(3)**.
3. $\{Inv$ and $H\}$ `skip` $\{Post\}$
   It is trivial since Inv and H is equivalent to the postcondition.
4. An appropriate variant is i: it is positive and strictly decreases at each iteration.

```
procedure SP1(A : in Array_Type; I : out Integer) ;
   --# derives I from A ;
   --# post 0 <=I and I < N
   --#     and (for all K in Index range  I+1.. N-1
   --#                              => A(K) >= A(K+1))
   --#     and ( I = 0 or A(I) < A(I + 1)) ;

procedure SP1(A : in Array_Type ; I : out Integer)
        is
         begin
            I:= N - 1 ;

           loop
            --#assert 0 <= I and I <= N - 1 and
            --# (for all K in Index range I+1..N-1
            --#                       => (A(K) >= A(K+1)));
            exit when I = 0 or else A(I) < A(I+1) ;

            I:= I - 1 ;
            end loop ;
        end SP1 ;
```

Figure 3.7: The Spark version of the subproblem SP1

Inserting the proofs in the Checker The verification conditions are generated from the corresponding Spark version depicted in Figure 3.7. Then, these verification conditions are submitted to the Simplifier which tries to evaluate them to true. We use the Checker to prove the verification conditions that the Simplifier has not succeeded to simplify to true.

Let us insert the manual proofs in the Checker: we can observe that adapting the manual proofs is not so natural: we have to know the several strategies that can be used by the Checker.

1. $\{a \text{ initialised}\}$ `i := n - 1 ;` $\{Inv\}$

   The manual proof of the first verification condition is simple. However, the Simplifier does not succeed in finding it: the simplified verification condition is the following

```
procedure_sp1_2.
H1:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 100 ->
       1 <= element(a, [i___1]) and element(a, [i___1]) <= 100)
       ->
C1:    for_all(k_ : integer, 100 <= k_ and k_ <= 99 ->
             element(a, [k_ + 1]) <= element(a, [k_]))
```

To prove `C1`, we do not need any hypothesis, the manual proof has shown that `C1` is true because there is no k such that $n \leq k < n$ (in the SPARK version `N` is determinated and is equal to 100). The goal is to lead the proof to conclude that one hypothesis is false, no `k_` exists.

The user eliminates the quantification by unfolding: `C1` becomes:

```
C1:   100 <= int_k__1 and int_k__1 <= 99 ->
      element(a, [int_k__1 + 1]) <= element(a, [int_k__1])))
```

which suggests a proof by implication: as the conclusion has the form $A \rightarrow B$, we insert A in the hypothesis, and B is now the conclusion.

```
H2:   100 <= int_k__1
H3:   int_k__1 <= 99
C1:   element(a, [int_k__1 + 1]) <= element(a, [int_k__1]))) .
```

We infer false from `H2` and `H3` which entails that this verification condition holds.

2. $\{Inv$ and not $H\}$ `i := i - 1;` $\{Inv\}$

```
procedure_sp1_3.
H1:   0 < i .
H2:   i <= 99 .
H3:   for_all(k_ : integer, i + 1 <= k_ and k_ <= 99 ->
          element(a, [k_ + 1]) > element(a, [k_]))) .
H4:   for_all(i___1 : integer, 1 <= i___1 and i___1 <= 100 ->
       1 <= element(a, [i___1]) and element(a, [i___1]) <= 100) .
H5:   i >= 0 .
H6:   element(a, [i + 1]) <= element(a, [i]) .
H7:   i >= 1 .
H8:   i <= 100 .
      ->
C1:   for_all(k_ : integer, i <= k_ and k_ <= 100 ->
         -> element(a, [k_ + 1]) >= element(a, [k_]))) .
```

In the manual proof, the assertion corresponding to `C1` is proved using two different hypotheses, **(2)** (corresponding to `H3`) and **(3)**(corresponding to `H6`), according to the value of the quantified variable. The strategy of this proof consists in proving by cases. We first remove the quantifier, we unfold and prove by implication and we obtain:

```
H9:    i <= int_k__1
H10:   int_k__1 <= 99 .
       ->
C1:    element(a, [int_k__1 + 1]) <= element(a, [int_k__1]) .
```

then, we prove by cases on `int_k__1 = i` or `int_k__1 > i`.

```
H11:   i = int_k__1 or i< int_k__1
```

- For `int_k__1 = i`,

  ```
  H12:   i = int_k__1
  ```

  `C1` becomes `element(a, [i + 1]) <= element(a, [i])` which is the hypothesis `H6`.

-
  ```
  H12:   i< int_k__1
  ```
  The case `int_k__1>i` is proved by unfolding `H3`.

  ```
  H13:  i + 1 <= int_K__1 and int_K__1 <= 99 ->
        element(a, [int_K__1 + 1]) <= element(a, [int_K__1])))
  ```

  We instantiate `int_K__1` with `int_k__1`.
  we infer `i+1 <= int_k__1` from `H12` (`H14`)
  we infer `C1` from `H14`, `H13`, `H10`.

3. $\{Inv$ and $H\}$ `skip ;` $\{Post\}$
   The corresponding verification condition is proved automatically.

4. The verification conditions for runtime checking are trivial for this algorithm, and are completely proved by the Simplifier.

5. The loop termination cannot be proved with the SPARK tools.

The proof of the main algorithm We present the interactive proof that the following algorithm computes the next permutation.

```
procedure Main(A : in out Array_Type ;
               C : in out Array_Type ; B : out Boolean)
       is
         I, J, T: Integer;
         begin

           SP1(A , I);

           if I > 0 then SP2(A , I , J);
                       T := A(J);
                       A(J) := A(I);
                       A(I) := T;
                       SP3(A,I);
                       B := true;

           else B := false;

           end if;
       end Main;
```

The interesting verification condition generated by the Examiner and simplified by the Simplifier is the following:

```
procedure_main_8.
H1:    perm1n(a) .
H2:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 100 ->
       1 <= element(a, [i___1]) and element(a, [i___1]) <= 100) .
H3:    for_all(i___1 : integer, 1 <= i___1 and i___1 <= 100 ->
       1 <= element(c, [i___1]) and element(c, [i___1]) <= 100) .
H4:    i__1 < 100 .
H5:    for_all(k_ : integer, i__1 + 1 <= k_ and k_ <= 99 ->
       element(a, [k_]) >= element(a, [k_ + 1])) .
H6:    element(a, [i__1]) < element(a, [i__1 + 1]) .
H7:    i__1 > 0 .
H8:    element(a, [i__1]) < element(a, [i__1 + 1]) .
H9:    i__1 < j__2 .
H10:   j__2 <= 100 .
H11:   element(a, [j__2]) > element(a, [i__1]) .
H12:   j__2 < 100 -> element(a, [j__2 + 1]) <= element(a, [i__1]) .
H13:   element(a, [i__1]) >= 1 .
H14:   element(a, [j__2]) <= 100 .
H15:   for_all(k_ : integer, i__1 + 1 <= k_ and k_ <= 100 ->
       element(a__3, [k_]) =
       element(update(update(a, [j__2], element(a, [i__1])), [i__1],
       element(a, [j__2])), [100 - (k_ - (i__1 + 1))])) .
H16:   for_all(k_ : integer, 1 <= k_ and k_ <= i__1 ->
       element(a__3, [k_]) =
       element(update(update(a, [j__2], element(a, [i__1])), [i__1],
       element(a, [j__2])), [k_])) .
H17:   for_all(i___1 : integer, 1 <= i___1 and i___1 <= 100 ->
       1 <= element(a__3, [i___1]) and element(a__3, [i___1]) <= 100) .
       ->
C1:    greater(a__3,a).
C2:    perm1n(a__3).
C3:    perm1n(c) and greater(c,a) -> greaterEq(c,a__3).
```

Except for some renamings, the verification condition is similar to the one we construct by symbolic execution.

Elaborating user-defined rules Before trying to make the proof, we define the several proof functions used in the specifications.

```
lexico(1): greater(X,Y) may_be_replaced_by
              for_some(p:integer,1 <= p and p <= 100 ->
                 for_all(q:integer,1 <= q and q < p ->
                        element(X,[q]) = element(Y,[q]))
                        and  element(X,[p]) > element(Y,[p])).
lexico(2): not greaterEq(X,Y) may_be_replaced_by greater(Y,X).
lexico(3): permut1N(X) may_be_replaced_by
```

```
               for_all(p:integer,1 <= p and p <= 100 ->
                       for_some(q:integer,1 <=q and q <= 100 ->
                           element(X,[q])= p)).
```

Proving C1 Before inserting the proof of `C1` in the Checker, let us first recall the structure of our manual proof attesting that $a \succ a_0$; we pick up in the proof detailed in Section 2.1.5.

> we prove that $\exists k : 1 \leq k \leq n : (\forall s : 1 \leq s < k : a[s] = a_0[s])$
> and $a[k] > a_0[k]$.
> We simply instantiate k with i:
> $(\forall s : 1 \leq s < i : a[s] = a_1[s] = a_0[s])$ by **(8, 11)**
> and $a[i] =_{(11)} a_1[i] =_{(9)} a_0[j] >_{(6)} a_0[i]$.

The hypotheses **(8)**,**(9)**, **(11)** that are used in this proof are represented in the hypotheses `H16` and `H9`. **(6)** corresponds to the hypothesis `H11`.
To insert this proof in the checker, we first replace `C1` by its definition (using `lexico(1)`):

```
C1:  for_some(s_ : integer, s_ >= 1 and s_ <= 100 and
               (for_all(k_ : integer, 1 <= k_ and k_ <= s_ - 1 ->
                element(tab__3, [k_]) = element(tab, [k_])
                )
               and element(tab__3, [s_]) > element(tab, [s_]))
             )
```

Then, we unfold this new `C1`; `C1` is replaced by four new goals:

```
  New goal C1: int_S__1 >= 1
  New goal C2: int_S__1 <= 100
  New goal C3: for_all(k_ : integer, 1 <= k_ and k_ <= int_S__1 - 1 ->
               element(tab__3, [k_]) = element(tab, [k_]))
  New goal C4: element(tab__3, [int_S__1]) > element(tab, [int_S__1])
```

We instantiate `int_S__1` by `i__1`, and `C1`, `C2` are proved immediately.

The subgoal `C3`: We use the strategy of unfolding and proving by implication; we obtain:

```
  New H19: 1 <= int_k__1
  New H20: int_k__1 <= i__1 - 1
  New goal C1: element(tab__3, [int_k__1]) = element(tab, [int_k__1])
```

Proving `C3` is reduced to prove the new goal `C1`: We unfold `H16` and we get:

```
  New H21: 1 <= int_K__1 and int_K__1 <= i__1 ->
          element(tab__3, [int_K__1]) =
          element(update(update(tab, [j__2], element(tab, [i__1])),
                        [i__1], element(tab, [j__2])),
                  [int_K__1])
```

We instantiate `int_K__1` with `int_k__1`
and we prove `C1` using `H20`, `H9` and `array(3)`:

```
array(3): element(update(A,J,X),K) & element(A,K) are_interchangeable
          if [J<>K] .
```

Indeed,

```
element(update(TAB, [i__1], element(tab, [j__2])), [int_k__1])
= element(TAB, [int_k__1])
```

where `TAB = update(tab, [j__2], element(tab, [i__1]))`
because `int_k__1 <> i__1` using `H20`

and

```
element( (update(tab, [j__2], element(tab, [i__1])),[int_k__1])
= element(tab, [int_k__1])
```

because `int_k__1 <> j__2` using `H9` and `H20`
So, `C3` is proved.

The subgoal `C4`: We use `H21`, we instantiate `K__1` with `i__1` and using `array(1)`

```
array(1): element(update(A,I,X),I) may_be_replaced_by X .
```

we get `element(tab__3, [i__1]) = element(tab, [j__2])`.
Using `H11`, `element(a, [j__2]) > element(a, [i__1])`;
by transitivity, `element(tab__3, [i__1]) > element(tab, [i__1])`.

By experimenting the Checker on this example, which proves that `a__3` is greater than `a`, one can admit that the user needs expertise and a lot of patience and perseverance to conclude his proofs. Anyway, the Checker allows one to make complex proofs such as for example the proof that `a__3` is the *next* permutation of `a`. Section 2.1.5 shows a tedious manual proof; it is possible to insert this in the Checker. We do not completely detail this proof which is very long, but the proof inserted in the Checker follows the same structure than the manual proof that is made by contradiction:

Proving C3

- First we consider that `permut1n(c) and greater(c,a~ )` are hypotheseses `H18, H19` resp. **(14)**, **(15)** in the manual proof.
- then we prove by contradiction: `not greaterEq(c,a)` becomes a hypothesis (`H20`) **(18)**.

- `not greaterEq(c,a)` can be replaced by `smaller(c,a)` and then by the quantified formula defining smaller. `greater(c,a~ )` must also be replaced by the definition.

  ```
  New H20: for_some(s1_ : integer, s1_ >= 1 and (s1_ <= 100 and
           (for_all(k1_ : integer, 1 <= k1_ and k1_ <= s1_ - 1 ->
                element(c, [k1_]) = element(a__3, [k1_])) and
                element(c, [s1_]) < element(a__3, [s1_]))))
  New H19: for_some(s2_ : integer, s2_ >= 1 and (s2_ <= 100 and
           (for_all(k2_ : integer, 1 <= k2_ and k2_ <= s2_ - 1 ->
                element(c, [k2_]) = element(a, [k2_])) and
                element(c, [s2_]) > element(a, [s2_]))))
  ```

- We make unfolding on the quantified hypotheses `H19` and `H20`.

  ```
  New H19: int_s2__1 >= 1 and (int_s2__1 <= 100 and
           (for_all(k2_ : integer, 1 <= k2_ and k2_ <= s2_ - 1 ->
               element(c, [k2_]) = element(a, [k2_])) and
               element(c, [s2_]) > element(a, [s2_]))))
  New H20: int_s1__1 >= 1 and (int_s1__1 <= 100 and
       (for_all(k1_ : integer, 1 <= k1_ and k1_ <= int_s1__1 - 1 ->
           element(c, [k1_]) = element(a__3, [k1_])) and
           element(c, [int_s1__1]) < element(a__3, [int_s1__1])))
  ```

Several times, we need the following rule:

```
lexico(4): for_all(k:integer, L <= k and k <= 100 ->
               for_some(q:integer,L <= q and
                   (q <=100 and element(a,[k]) = element(a2,[q]))
                       )
                  )
           may_be_deduced from
[perm1N(a);
 for_all(s:integer, 1 <= s and s <= L-1
                 -> element(a,[s])=element(a2,[s]))]
```

We should have proved this before proving the verification condition that we consider. Notice that in the manual proof of Section 2.1.5, we do not detail this proof either.

- We prove `C1` by cases: for each case we have to find a contradiction between hypotheses. This structure corresponds exactly to the way we prove it manually.
  - `int_s2__1 < i__1`
  - `int_s2__1 > i__1`
  - `int_s2__1 = i__1`

    Replacing all `int_s2__1` by `i__1`, we prove by cases on:
    * `int_s1__1 = i__1`
    * `int_s1__1 < i__1`
    * `int_s1__1 > i__1`

3.2.3 Conclusion

In the context of learning to construct programs based on invariants and decomposition into subproblems, SPARK is interesting because it follows the principles of structured programming. To check the correctness of a code fragment involving a loop, we only have to give its specification and loop invariant. The verification conditions are generated and verified independently from each other. Also, for verifying code fragments involving subproblems, SPARK only uses the specifications of the subproblems. Notice however that there is no straightforward way to provide a variant to prove loop termination.

Drawbacks SPARK is a well designed tool but it suffers from three limitations in our specific context.

- A first limitation is that, very often, the SPADE Simplifier is not powerful enough to finish proofs. Besides, when the proof has not succeeded, it is not clear whether it is because the verification condition does not hold, or because the tool needs additional rules, or even because Simplifier is not powerful enough.

- The second limitation is that the user has to analyse the failure manually. The tool does not provide counter-examples that could highlight the cause of failure.

- The third limitation lies in the proof checker, which requires too much expertise from the user in our context of learning. The user has to spend a lot of time to learn to manipulate the tool, and to develop correctness proofs.

Actually, SPARK could be used in a more advanced context than ours. Its assertion language is expressive and its Ada subset is a pedagogical language. Since the user can visualise generated verification conditions and log files of automatic proofs, the tool could be useful in a follow-up course devoted to formal verification of software.

As ESC/Java2, SPARK is not able to provide counter-examples, with concrete values, attesting that the algorithm is not correct according to its specifications. In the next section, we present SMV, a tool able to provide counter-examples and to pinpoint the errors.

3.3 SMV

3.3.1 Overview

To check the correctness of systems, formal methods offer opportunities for mechanical verification, but most existing techniques either require extensive human guidance, or are limited to verify simple properties, in term of verification they are generally unsound and/or incomplete. Using tools based on formal methods involving theorem provers has shown these disadvantages in our pedagogical context.

Symbolic model checking (SMV [44]) on Binary Decision Diagrams is an efficient automatic verification technique that is simultaneously capable of scaling and of verifying a wide range of properties. It has been applied successfully to many industry-scale hardware circuits and protocols; but in our context of learning a programming methodology, SMV could also be appropriate as it is fully automatic and it generates concrete counter-example when a property does not hold.

3.3.2 Model Checking

Model checking is a formal-verification technique based on state exploration [5]. Given a state transition system and a property, model checking algorithms exhaustively explore the state space to determine whether the system satisfies the property. The result is either a claim that the property is true or else a counter-example (a sequence of states from some initial state) falsifying the property.

3.3.2.1 CTL Model Checking

In temporal-logic model checking, we are given a state transition system, which models a software or hardware system, and a property specified as a formula in a certain temporal logic, and we determine whether the system satisfies the formula. A common logic for model checking is the branching time Computation Tree Logic CTL, which extends propositional logic with certain temporal operators.

Formally, a state transition system $\langle Q, R, I \rangle$ consists of a set of states Q, a state transition relation $R \subset Q \times Q$, and a set of initial states $I \subseteq Q$. A *path* is an infinite sequence of states such that each consecutive pair of states is in R. For simplicity, we discuss just a subset of CTL, namely the subset with only the temporal operators AG, AF, EG and EF. We say that a *proposition* is any Boolean combination of predicates on the state variables.

[5]This subsection is strongly inspired from [9]

A *formula* is either a proposition, a Boolean combination of formulas, or of the form AG f, AF f, EG f, or EF f where f is a formula. Each formula is evaluated at some state q. A proposition holds at q if q satisfies the proposition. The operator A means "for all paths starting at q", E means "for some path starting at q", G means "for every state along the path" and F means "for some state along the path. So AG *safe* holds at q if every state (G) along every path (A) starting at q satisfies the proposition *safe*. The system satisfies a formula if the formula holds at all initial states. If not, a model checker attempts to find a counter-example. For example, if the formula AG *safe* is false, a counterexample is a finite path starting at some initial state and ending at a state that is not *safe*.

3.3.3 Symbolic Model Checking

In explicit model-checking techniques, the truth value of a CTL formula is determined in a graph-theoretic manner by traversing the state diagram, with time complexity linear in the size of the state space and the length of the formula. Unfortunately, the size of the state space is often exponential in the size of the system description, resulting in the state explosion problem.

An important breakthrough in model checking was the introduction of symbolic techniques. Instead of visiting individual states, symbolic model checking visits a set of states at a time. A state set can be represented by a predicate on the state variables such that a state is in the set if and only if the predicate is true in the state. The efficiency of symbolic model checking relies on succinct representations and efficient manipulations of these predicates.

3.3.3.1 Binary Decision Diagrams

When the state space is finite, we can assume without loss of generality that the state variables are Boolean and there are only finitely many of them. A predicate on these variables is simply a Boolean function, which can be represented by reduced ordered binary decision diagrams (BDDs).

Intuitively, a BDD is like a binary decision tree, except that isomorphic subtrees must be combined resulting in a directed acyclic graph. In addition, each path can contain a variable at most once, and must comply with a fixed linear order of the variables. BDDs are canonical and Boolean operations such as conjunction, disjunction and negation can be computed in polynomial time. BDDs are usually small, but often their sizes depend critically on the variable order.

3.3.4 SMV

SMV is a CTL symbolic model checker using BDDs to represent state sets and transition relations. In SMV, 1 represents *true* and 0 represents *false*. The logical operators `and`, `or` and `not` are `&`, `\` and `!`, respectively. An SMV program consists of a description of a finite-state transition system and a list of CTL formulas. Recall that a transition system is defined by a state space, a transition relation and a set of initial states.

The state space is determined by state variable declarations, preceded by the keyword `VAR`. For example,

```
VAR
   b: boolean;
   x: 0..7;
   state: {inv, post}
```

declares a Boolean variable `b`, a integer variable ranging between 0 and 7, and a variable `state` with value drawn from the set `inv, post`. The variable `x` is internally represented as three Boolean variables.

The transition relation and the inital states can be specified by a collection of simultaneous assigments: Initial-state assignments are made simultaneously at the start, and subsequently next-state assignments are simultaneously executed one per cycle. Assigments are preceded by the keyword `ASSIGN`. For any variable *var*, init(var) refers to the value of var in the initial states, so the code

```
ASSIGN
   init(b) := 0;
```

sets the initial values of `b` to 0 (i.e., false). To define the transition relation, the expression `next(var)` represents the value of *var* in the next states. Therefore,

```
ASSIGN
   next(x) := case state = inv & b = 0: x + 1 ;
                   1: 0 ;
              esac
```

specifies `x` will be incremented by 1 in the next state if we are in state `inv` and value of `b` is currently 0.

An alternative way to specify the transition relation is to use the keyword `TRANS`, followed by an arbitrary expression involving the state variable, defined symbols and/or their next versions. The expression directly defines the transition relation as a proposition. The following `next(state)` above is equivalent to the following:

```
TRANS
 (state = inv & b = 0) -> next(x) = x+1
```

Next-state assignments define the transition relation imperatively, whereas `TRANS` statements define it declaratively. `TRANS` are however less robust; for example, an empty transition relation can be specified with `TRANS` statements, resulting in strange analysis results. Such problems can be hard to track down, so `TRANS` statements must be used with care. A program can contain both next-state assignments and `TRANS` statements. Their conjunction forms the transition relation.

The Cadence SMV model checker [43] is an evolution of the original SMV [44]: it allows several forms of specification, including the temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications. It also includes a graphical user interface.

3.3.5 Supporting the Methodology with SMV

To get a representative idea of the possibilities of SMV in our pedagogical context, we propose to verify the insertion sort algorithm that we have constructed and proved in Section 2.1.4. This manipulates arrays, includes decomposition into subproblems and non trivial predicates. To have it right in front of us, we recall the main algorithm with its specification in Figure 3.8 as well as the subproblem called specification.

Modelling the Statements in SMV

We first represent each part of every subproblem as a transition system, where states correspond to key program points (at least the initial and final points) and transitions are sequences of assigments. A conditional statement is translated into two conditional transitions. A call to a subproblem can be translated into one transition from a precondition state to a postcondition state.

As an example, we write the script partially shown in Figure 3.9 to translate the statement `SP; i+1` (i.e., the statement *Iter* of Figure 3.8).

- We define four states. The states `state-iter1` and `state-iter2` correspond to the program points before and after the iteration.

- Declarations:
  const n ;
  tab a : array $[1..n]$ of integer; (input, output)
  var i : integer; (auxiliary variable)
- Pre: a initialised and $n \geq 0$
  Post: $(\forall j : 1 \leq j < n : a[j] \leq a[j+1])$ and a is a permutation of a_0
  Inv: $0 \leq i \leq n$ and a is a permutation of a_0 and
  $(\forall k : 1 \leq k < i : a[k] \leq a[k+1])$ and
  $a[i+1..n]$ is unchanged
- Init: `i := 0 ;`
  Halting condition: `i != n`
  Iter: `SP ; i := i + 1`
- SP specification:
  Pre: $0 \leq i < n$ and $(\forall k : 1 \leq k < i : a[k] \leq a[k+1])$
  Post: $(\forall k : 1 \leq k \leq i : a[k] \leq a[k+1])$
  and a is a permutation of a_0
  and $a[i+2..n]$ is unchanged

Figure 3.8: The main subproblem of the insertion sort algorithm

The states `state-preSP` and `state-postSP` represent the program points before and after the call to the subproblem.

- The `ASSIGN` declaration defines the next value of i depending on its value in the previous state. It also expresses that `a` is unchanged except during the call to the subproblem.
- The `TRANS` declaration expresses how the value of a can be modified by the call to the subproblem using a relation between its value before and after the call. (We explain later in this Section how pre/post conditions can be translated to the corresponding conditions `assert-preSP` and `assert-postSP`.)

It should be noticed that the array size is not parameterised and that, for each variable (even an auxiliary variable), the user has to choose adequate finite domains.

Translating the assertions to SMV Translating assertions into SMV is possible as we work on finite domains (although formulas can become quite complicated). As an example, let us show how we can express that $a[1..i]$ is a permutation of $a_0[1..i]$. In fact, it is not possible to express a relation between the initial and final values of `a` without modifying the

```
MODULE main
VAR
  a : array 1..4 of  1..4;
  i: 0..5;
  state : {state-iter1,state-iter2,state-preSP,state-postSP};
ASSIGN
--relations between program points
  next(state):= case state = state-iter1 & i < 4 : state-preSP;
                     state = state-preSP : state-postSP;
                     state = state-postSP : state-iter2;
                     1: state;
                esac;
--i:=i+1 in Iter
  next(i):= case state=state-postSP &i<5: i+1;
                 1:i;
            esac;

--a[i] is unchanged unless the SP is called
  next(a[1]):= case state = state-preSP : 1..4;
                     1: a[1];
               esac;
  next(a[2]):= case state = state-preSP : 1..4;
                     1: a[2];
               esac;
  next(a[3]):= case state = state-preSP : 1..4;
                     1: a[3];
               esac;
  next(a[4]):= case state = state-preSP : 1..4;
                     1: a[4];
               esac;
--SP call
  TRANS
     (state= state-preSP & assert-preSP) -> (assert-postSP)
```

Figure 3.9: Transition system corresponding to the main problem iteration

scripts for the statements. We need to introduce a new array `a0` and to specify that it is left unchanged by every transition[6]. Then, with the help of several intermediate modules, we can translate the assertion as an instance of the module `perm(a,b,i)`. Note that the `result` fields of the intermediate modules are useful to pinpoint the violated parts of an assertion when a counter-example appears.

```
MODULE count(a,x,i)
-- #x dans a[1..i]
DEFINE result := (a[1]=x & 1<=i) + (a[2]=x & 2<=i)
                + (a[3]=x & 3<=i) + (a[4]=x & 4<=i);

MODULE samecount(a,b,x,i)
-- #x dans a[1..i]=? #x dans b[1..i]
VAR count_a : count(a,x,i);
    count_b : count(b,x,i);
DEFINE result:= count_a.result = count_b.result;

MODULE perm(a,b,i)
VAR samecount1 : samecount(a,b,a[1],i);
    samecount2 : samecount(a,b,a[2],i);
    samecount3 : samecount(a,b,a[3],i);
    samecount4 : samecount(a,b,a[4],i);
DEFINE result:= samecount1.result & samecount2.result
               & samecount3.result & samecount4.result;
```

Quantified assertions such as $(\forall j : 1 \leq j < i : a[j] \leq a[j+1])$ or "$a[i+1..n]$ is unchanged" can be defined similarly, using other modules instantiated in the main module. Writing these modules is tedious because all correct valuations of the variables must be enumerated:

```
MODULE increasing(a,i)
DEFINE result:= (i=0) | (i=1) | (i=2 & a[1]<=a[2]) |
                (i=3 & a[1]<=a[2] & a[2]<=a[3]) |
                (i=4 & a[1]<=a[2] & a[2]<=a[3] & a[3]<=a[4]);
MODULE unchanged(a,a0,i)
DEFINE result:= (i=4) | (i=3 & a[4]=a0[4]) |
            (i=2 & a[4]=a0[4] & a[3]=a0[3]) |
            (i=1 & a[4]=a0[4] & a[3]=a0[3] & a[2]=a0[2]) |
            (i=0 & a[4]=a0[4] & a[3]=a0[3] & a[2]=a0[2] & a[1]=a0[1]);
```

Defining `increasing_a`, `unchanged_a` and `perm_a_a0` as instances of the modules `increasing`, `unchanged` and `perm`, we can finally formalise the following assertions:

```
DEFINE assert-preSP :=  0<=i & i< 4 & increasing_a.result;
```

[6]The whole script is in Appendix A.3.1

```
--0<=i<4 && forall j:1<=j<i: a[j]<=a[j+1]

DEFINE assert-postSP:= (unchangedN_a.result  & increasingN_a.result
                              & perm_a_a0N.result
                       );

DEFINE assert-iter1:= i<4 & increasing_a.result & unchanged_a.result
                      & perm_a_a0.result; --Inv and not H
DEFINE assert-iter2:= i<=4 & increasing_a.result & unchanged_a.result
                      & perm_a_a0.result; --Inv
```

Checking the correctness of the program with SMV

Given the whole script, which is in Appendix A.3.1, we can check the proposition

$$\{Inv \text{ and not } H\}\ iter\ \{Inv\}$$

. It can be translated into temporal logic formulas as follows:

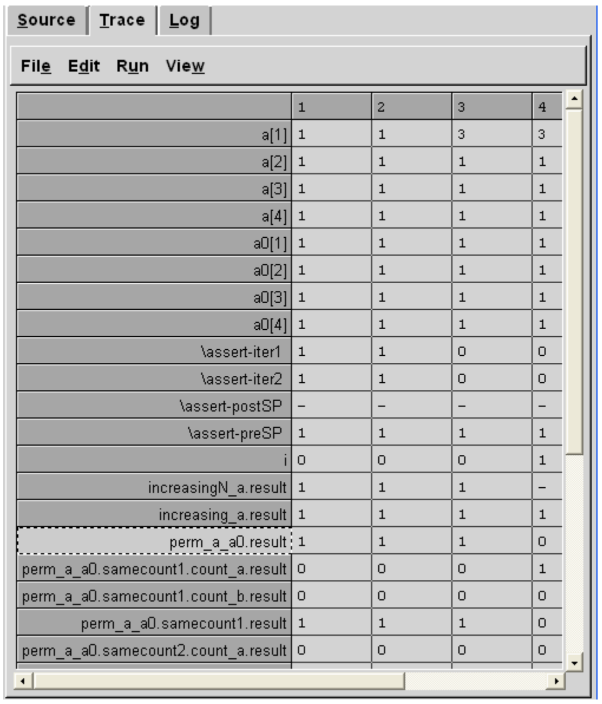
```
SPEC ((state=state-iter1) & assert-iter1) -> AF(state = state-iter2)
SPEC ((state=state-iter1) & assert-iter1) ->
     AG (state = state-iter2 -> assert-iter2)
```

The first formula states that the statement *iter* always terminates. The second formula says that the invariant *Inv* holds after executing this statement if the condition $\{Inv$ and not $H\}$ holds beforehand.

With the variable domains given in Figure 3.9, Cadence SMV is able to prove the correctness of $\{Inv$ and not $H\}$ *iter* $\{Inv\}$ in 20 ms. The subproblem is proved in 40 ms.

The feedback In case of any mistake in the (formalisation of the) program and of its assertions, counter-examples are provided by SMV. For example, if the subproblem postcondition is not strong enough; we forget to say that $a[1..i+1]$ is a permutation of $a_0[1..i+1]$, Cadence SMV gives a counter-example that we can interpret in Figure 3.10:
first column gives a variable state satisfying $\{Inv$ and not $H\}$ (`assert_iter1`). Columns 2 and 3 represent the variable states before and after the subproblem call; the last column gives a variable state that does not satisfy the invariant (`assert_iter2`). If appropriate modules are defined, errors can be pinpointed: one can observe, in the fourth column, the violation of `perm_a_a0.result` saying that `a[1..i]` is a permutation of `a_0[1..i]`.

Loop termination With SMV, it is possible to check termination by brute force: we can write the transition system corresponding to the complete algorithm and ask SMV to verify the temporal logic property that,



	1	2	3	4
a[1]	1	1	3	3
a[2]	1	1	1	1
a[3]	1	1	1	1
a[4]	1	1	1	1
a0[1]	1	1	1	1
a0[2]	1	1	1	1
a0[3]	1	1	1	1
a0[4]	1	1	1	1
\assert-iter1	1	1	0	0
\assert-iter2	1	1	0	0
\assert-postSP	–	–	–	–
\assert-preSP	1	1	1	1
i	0	0	0	1
increasingN_a.result	1	1	1	–
increasing_a.result	1	1	1	1
perm_a_a0.result	1	1	1	0
perm_a_a0.samecount1.count_a.result	0	0	0	1
perm_a_a0.samecount1.count_b.result	0	0	0	0
perm_a_a0.samecount1.result	1	1	1	0
perm_a_a0.samecount2.count_a.result	0	0	0	0

Figure 3.10: The Cadence SMV feedback

using data satisfying the precondition, every execution will lead to the post-condition state: `assert-pre1 -> AF(state = state-post)` [7]. Nevertheless, in order to support the programming methodology, we must use another method and write a new script modelling $\{Inv$ and not B and $v_0 = variant\ \}\ Iter; v := variant\ \{v < v_0$ and $v \geq 0\}$.

3.3.6 Conclusion

SMV is powerful enough to represent the kind of imperative programs we deal with and their associated verification conditions. We can express and check all desirable properties; translating assertions into SMV is possible as we work on finite domains.

SMV is sound and complete in the restricted variable domains. It is able to give counter-examples, with concrete values that violate a property; it can also pinpoint violated parts of the property checked. This tool is very efficient in our context, but time is increasing with the size of the variable domains. No matter, since we can consider that, if an algorithm is correct for small size arrays, it is most probably correct for large size arrays, and if an algorithm is correct for small variable domains (well chosen), it is most

[7]The state `state-post` corresponds to the last program point.

probably correct considering the same variables with non restricted integer domains. The way of formalising assertion has no impact on the completness or soundess of a verification. It can just have an impact on efficiency. However, the work that is required to formalise these properties is well beyond what can be asked to students learning the methodology. Besides, many mistakes can simply be made when they translate the statements in transition system.

It is clearly possible to extend SMV with a higher level language, in order to get a tool translating algorithms into transition systems and translation expressive assertions into CTL properties. This could possibly be done using a macro preprocessor like M4 [4].

Comparing with the verification tools based on theorem proving, it appears that making an exhaustive verification such as SMV does is appropriate since our objective is to be as precise as possible, and to give counter-examples for helping the user to understand his errors.

3.4 Software Model Checkers

In the previous section, we have concluded that making an exhaustive verification to get a precise feedback about the correctness of our algorithms seemed the best approach even if we have to restrict the data domains to small finite domains. Using SMV, we have observed the difficulties to translate our algorithms and specifications into transition systems and CTL properties. An alternative to a pure model checker like SMV could be the use of software model checkers. Problems of translation into transition systems are no more to be considered and these tools use model-checking techniques to "check the correctness" of the algorithms. We consider *Blast* [27], *Java PathFinder* [5], *BANDERA* [13] and*CBMC* [11].

The *Blast* [27] tool is a software model-checker that focuses on checking sequential C code, using well-engineered predicate abstraction and abstraction refinement tools. The abstraction is made on the fly; a first model is model-checked: if there are no wrong path, the model is safe. Otherwise, we check the counter-example using symbolic execution. If it is non relevant, the abstraction is refined. Assertion expressiveness is restricted to the C syntax and Blast does not deal with arrays up to now. We tried to use it for algorithms that do not use arrays like indian exponentiation, simple integer division, and computing x^2 using the relation $(x+1)^2 = x^2+2x+1$. Problems appear probably because the theorem prover used by Blast does not handle full arithmetic.

Java PathFinder [5] is a system to verify Java byte code: it systematically explores all potential execution path. JML [7] specifications can partially be checked, in fact they are executed/evaluated thanks to JMLC [10]. The verification is not compositional: no invariant is used and the verification of a problem does not use the specification of the called subproblems; it uses their code. So, *JPF* cannot enforce the use of the structured programming method. Besides, the verification is not complete since only a subset of JML assertions can be executed (like with ESC/Java2).

BANDERA is a software to analyse Java source code, interfacing it with verification tools like JPF, SPIN [33], SMV. However, these tools do not seem appropriate for our purpose, because they do not focus on specifications but on the code. The verification cannot be compositional.

CBMC [11] is a bounded model-checker fo C programs, its assertion language is not expressive enough, the verification is not compositional: it does not need any loop invariant because it unrolls loop.

These tools have not been deeply investigated, but, to our knowledge, they are limited to handle specific properties of programs, not fully complete specifications. Their objective is different. Since they have to check programs written in a 'real life' language, they must restrict to specific properties such as checking out of bounds errors, for instance.

3.5 Conclusion

In conclusion, we observe that existing tools have never the same objectives as ours. They focus on real life programs, using a complex language. They focus on some specific properties; their goal is not to give a complete and sound verification according to the specification. They do not want to enforce the use of the decomposition into subproblems and the use of the invariant; on the opposite, they try to do not disturb the programmer behaviour.

With this observation, we have built our own tool. Programs, specifications, variants, invariants can be expressed straightforwardly; the verification is compositional. Moreover, it is able to produce concrete counter-examples. The next chapter presents this tool.

Chapter 4

MPVS: A New Tool to Teach Structured Programming

We are convinced that students, and still more computer science students, are motivated when they use computer tools; the difficulty is to provide a very well adapted tool according to our pedagogical objectives. We have searched for an existing tool that could help our students to elaborate their programs with the structured programming method, but all the tools analysed are not adapted: using verification tools like ESC/Java2 or Spark, the student cannot get a precise feedback: too often ESC/Java2 gives false warnings, and sometimes it forgets errors; too often Spark is not powerful enough to finish the proofs. These tools do not provide concrete counter-examples and, even when they are said to be automatic, they require too much effort and too much expertise from the user to be able to manipulate it. SMV has the advantage to be automatic and to give precise feedback to the user but it cannot be used as it is because the work required to translate an algorithm with its specification is well beyond what can be asked to students learning the methodology. In a general way, all these tools requires more notions than what we want to teach; if using a tool should help the student to easily understand the concepts to be learnt, this objective may not be reached when the tool is not appropriate.

In our programming learning context, we would like a fully automatic tool, easy to manipulate, enforcing the use of the structured programming method and giving precise feedback to help the students to understand their reasoning errors. We do not want to consider that students need any notion of theorem provers or model checkers. Students would just need to formalise assertions and to write code, since these are the goals of our programming course. The ideal tool should be easy to use, while using the non trivial taught concepts, this should show the witchery of the method, while enforcing rigor, the role of the loop invariant and the decomposition into subproblems. The feedback given by the tool should be clear, to help our students to understand their reasoning errors. Displaying precise counter-examples may be very informative and false warnings and forgotten errors have to be avoided.

We have designed and implemented a tool that fulfils those requirements better, in our opinion. The presentation of the tool, that we name MPVS has been published in [21]. MPVS considers a very simple programming language which handles naturals and Booleans. The assertion language mainly contains quantifiers over finite domains, arbitrary arithmetic relations and also some predicates that we have considered useful by experimentation. The concrete syntax of the assertion and the programming languages are fully described in Appendix B.1. We now present what our tool can do through the several examples for which we have already tried to check the correctness using Spark, ESC/Java2 or SMV.

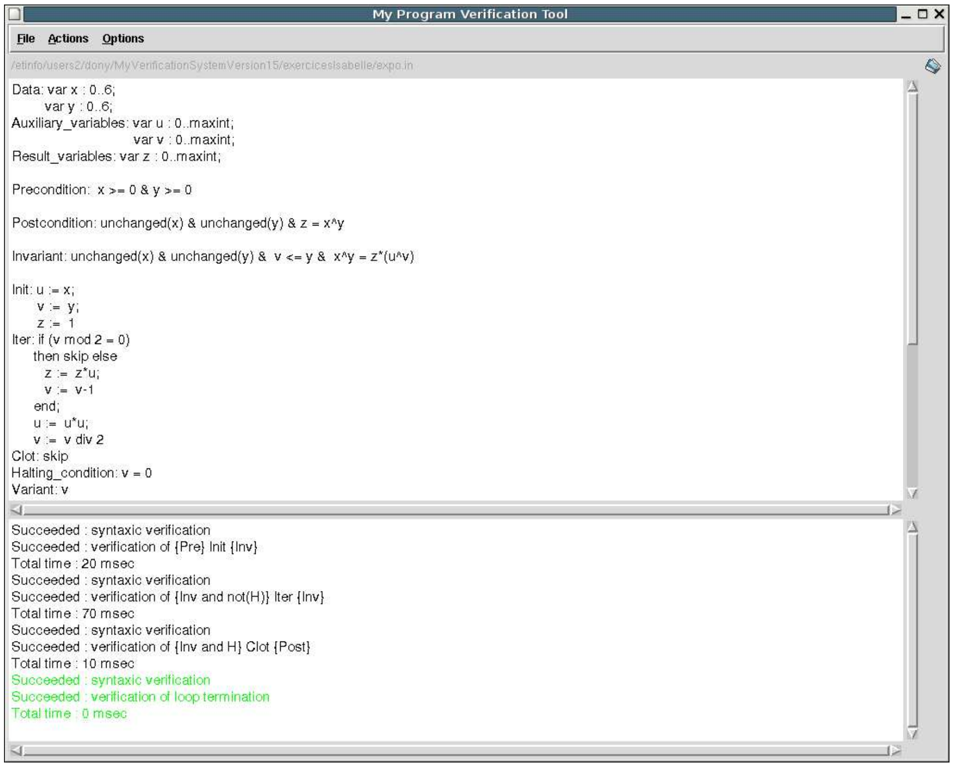
4.1 What our Tool can Do

4.1.1 The Indian Exponentiation

To describe the several functionalities of MPVS, let us first have a look at Figure 4.1 that shows concretely a picture of our tool when it is used to check the correctness of the Indian exponentiation algorithm. The program is written in a specific format in an editor provided by the tool. Above this editor, there is a textfield with the name of the file (if it exists) corresponding to the text displayed on the editor. On the right, we have a button providing the list of opened files to easily go from one file to another. The second text area gives feedback on the verification, we name this the feedback area. Each opened file has its own editor and feedback area.

The menu contains three items: *Files* allows one to open, close and save files; *Actions* allows the verification operations.

Syntax and syntactic verification Figure 4.2 displays the algorithm together with its specifications, exactly as it must be written in the tool editor. This example only uses integer variables. Let us make it precise that `x^y` stands for x^y and that `unchanged(x)` expresses that `x` is unchanged with respect to its value at the beginning of the program execution. A loop is explicitly decomposed into the initialisation (`Init`), the iteration (`Iter`), the closure (`Clot`) and the halting-condition (`Halting_condition`). Besides, to prove termination of the loop, the user can provide a variant. MPVS first verifies the syntax of the algorithms and specifications. To give a clear feedback to the user, syntactic errors are underlined in the text editor.


My Program Verification Tool
File Actions Options
/etinfo/users2/dony/MyVerificationSystemVersion15/exercicesIsabelle/expo.in

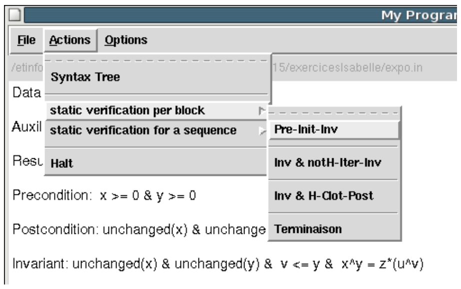
```
Data: var x : 0..6;
      var y : 0..6;
Auxiliary_variables: var u : 0..maxint;
                     var v : 0..maxint;
Result_variables: var z : 0..maxint;

Precondition:  x >= 0 & y >= 0

Postcondition: unchanged(x) & unchanged(y) & z = x^y

Invariant: unchanged(x) & unchanged(y) &  v <= y &  x^y = z*(u^v)

Init: u := x;
    v := y;
    z := 1
Iter: if (v mod 2 = 0)
    then skip else
      z := z*u;
      v := v-1
    end;
    u := u*u;
    v := v div 2
Clot: skip
Halting_condition: v = 0
Variant: v
```

```
Succeeded : syntaxic verification
Succeeded : verification of {Pre} Init {Inv}
Total time : 20 msec
Succeeded : syntaxic verification
Succeeded : verification of {Inv and not(H)} Iter {Inv}
Total time : 70 msec
Succeeded : syntaxic verification
Succeeded : verification of {Inv and H} Clot {Post}
Total time : 10 msec
Succeeded : syntaxic verification
Succeeded : verification of loop termination
Total time : 0 msec
```

Figure 4.1: The graphical interface

```
Data: var x : 0..6;
      var y : 0..6;

Auxiliary_variables: var u : 0..maxint;
                     var v : 0..maxint;

Result_variables: var z : 0..maxint;

Precondition: x >= 0 & y >= 0

Postcondition: unchanged(x) & unchanged(y) & z = x^y

Invariant: unchanged(x) & unchanged(y) & v <= y & x^y = z*(u^v)

Init: u := x ;
      v := y ;
      z := 1

Iter: if (v mod 2 = 0)
      then skip else
         z := z * u ;
         v := v - 1
      end;
      u := u * u ;
      v := v div 2

Clot: skip

Halting_condition: v = 0

Variant: v
```

Figure 4.2: The Indian Exponentiation algorithm with its specification



Precondition: x >= 0 & y >= 0

Postcondition: unchanged(x) & unchange

Invariant: unchanged(x) & unchanged(y) & v <= y & x^y = z*(u^v)

Figure 4.3: The verification menu

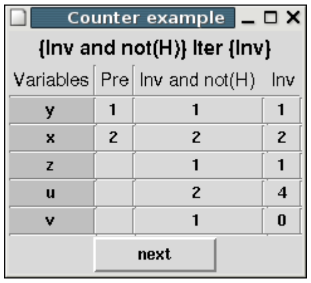
Checking the correctness of an algorithm To enforce the verification method and the role of the loop invariant, MPVS concretely forces the user to verify each Hoare proposition independently as we can observe in the item unrolled in Figure 4.3.

Let us analyse the behaviour of the tool when it checks a Hoare proposition. Since the proposition { *Pre* } *Init* { *Inv* } is correct, the tool simply displays, in the feedback area, the following message (see Figure 4.1):

```
Succeeded :  verification of {Pre} Init {Inv}
```

Similar messages are displayed for the other Hoare propositions. Since each Hoare proposition and termination are proved to be correct, the user can deduce that his algorithm is correct according to its specification. Using a AMD Ahtlon XP 2800+, $2GHz$ CPU with $1GB$ RAM, our system is able to prove the correctness of the algorithm 4.2 in 100 ms (adding up the execution times of all Hoare propositions).

Generating counter-examples Let us have a look at the tool behaviour if the user makes an error in the code, for example, he forgets to write the assignment `z := z * u` in the conditional statement where `v mod 2 = 1`. When verifying { *Inv* and not *H* } *Iter* { *Inv*}, the tool detects some problems after 30 ms. A message is printed in the feedback area with a counter-example; this counter-example is also presented in the following table.

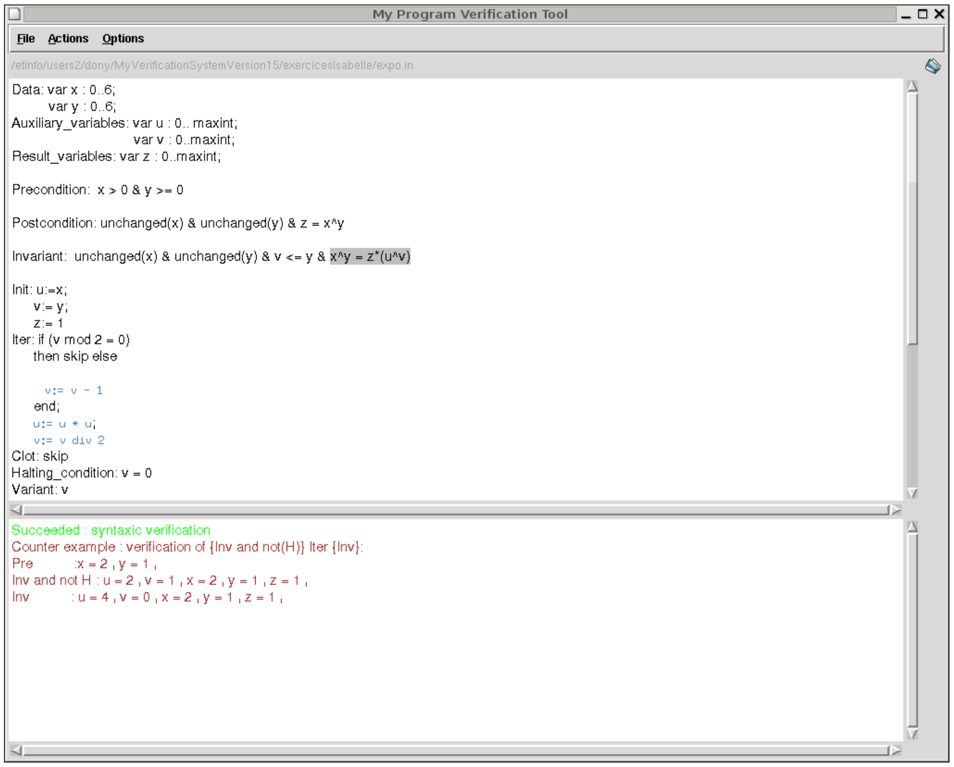


{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
y	1	1	1
x	2	2	2
z		1	1
u		2	4
v		1	0

Other counter-examples can be displayed thanks to the `next` button. This table gives an example of input values satisfying the precondition (column 2), column 3 displays a state satisfying the invariant { *Inv* and not *H* } before an execution of the iteration and the last column provides the state obtained after executing the iteration. It can be observed that the assertion { *Inv* } is false in this state.

In addition, the violated part of the invariant and the involved sequence



```
/etinfo/users2/dony/MyVerificationSystemVersion15/exercicesIsabelle/expo.in
```

```
Data: var x : 0..6;
        var y : 0..6;
Auxiliary_variables: var u : 0.. maxint;
                              var v : 0..maxint;
Result_variables: var z : 0..maxint;

Precondition:  x > 0 & y >= 0

Postcondition: unchanged(x) & unchanged(y) & z = x^y

Invariant:  unchanged(x) & unchanged(y) & v <= y & x^y = z*(u^v)

Init: u:=x;
     v:= y;
     z:= 1
Iter: if (v mod 2 = 0)
     then skip else

       v:= v - 1
     end;
     u:= u * u;
     v:= v div 2
Clot: skip
Halting_condition: v = 0
Variant: v
```

```
Succeeded : syntaxic verification
Counter example : verification of {Inv and not(H)} Iter {Inv}:
Pre          :x = 2 , y = 1 ,
Inv and not H : u = 2 , v = 1 , x = 2 , y = 1 , z = 1 ,
Inv          : u = 4 , v = 0 , x = 2 , y = 1 , z = 1 ,
```

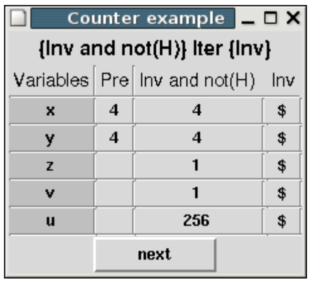
Figure 4.4: The tool feedback for a wrong version of the Indian exponentiation algorithm

of statements are underlined in the text as we can see in Figure 4.4.

Out of domain errors The tool can also detect out of bound of variable domain. For example, if we maintain the original variable domains except for the variable `u` where we restrict its domain to `0..46656` (6^6), the tool pinpoints nearly instantly a runtime error; the feedback is the following:

```
Problem in {Inv and not(H)} Iter {Inv} :
The value 65536 is out of the domain of the variable u

Precondition :  x = 4 , y = 4
Invariant :  u = 256 , v = 1 , x = 4 , y = 4 , z = 1
Program point :  u = 256 , v = 0 , x = 4 , y = 4 , z = 256
```

The third variables state (`Program point`) corresponds to the program state just before the underlined assignment `u := u * u` which is involved in the runtime error.



Counter example

{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
x	4	4	$
y	4	4	$
z		1	$
v		1	$
u		256	$

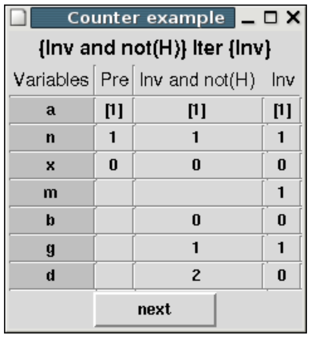
This table gives an example of input values satisfying the precondition (column 2), column 3 displays a state satisfying the invariant $\{Inv$ and not $H\}$ before an execution of the iteration; the execution of the iteration will generate a runtime error before reaching the program point where the invariant must hold as it is represented in the last column.

4.1.2 Binary Search Algorithm

Now, we consider again the binary search algorithm that we have elaborated and manually proved in Section 2.1.3 and that we have automatically proved with ESC/Java2 in Section 3.1. We show the ability for our tool to manage arrays, Boolean, existential and universal quantifiers on integer variables, implication, equivalence, evaluations from left to right. Let us look at Figure 4.5. The precondition expresses that `x` is initialised and that the array `a` is sorted. It is important to notice that, contrary to ESC/Java2, the way of formalising specifications has no consequence on the proof of correctness. We could have written `(forall i:1 <= k <= n:(forall i:  1 <= l <= n:k < l => a[k] <= a[l]))` instead. In postcondition, we use the predicate `unchanged(1, n:a)` to say that the array `a` is not modified; we use the operator of equivalence to express the role of the result variable `b`. In the invariant, we use a logical operator `&&` which evaluates assertions from left to right; we consider that, in an assertion, we cannot have expressions that are not well defined. For examples, out of bound array index and division by zero are prohibited in an assertion as they are prohibited in a statement. With these considerations, assertions must sometimes be evaluated in a specific order.

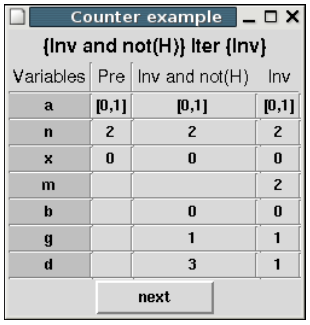
Proving the algorithm correctness Our system is able to prove the correctness of this program in 10ms (for $n = 0$), 40ms ($n = 1$), 120ms ($n = 2$), 710ms ($n = 3$), 5.34s ($n = 4$), and 61.72s ($n = 5$). But we do not have to fix the array size: we can choose, as in Figure 4.5, a finite domain for `n`, instead. Indeed, it may be useful to automatically verify borderline cases involving arrays (cases where $n = 0$ and $n = 1$). It takes 1min 6.4s to prove the correctness of the program for all n in [0..5].

Finding errors in an incorrect binary search algorithm Now, we keep the same specification and loop invariant, but we introduce an error in the algorithm: we replace the statement `d := m` by `d := m - 1`. the system gives the following counter-examples.


Counter example

{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
a	[1]	[1]	[1]
n	1	1	1
x	0	0	0
m			1
b		0	0
g		1	1
d		2	0


Counter example

{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
a	[0,1]	[0,1]	[0,1]
n	2	2	2
x	0	0	0
m			2
b		0	0
g		1	1
d		3	1

The first table displays the first counter-example which is found after 10 ms. The second table displays the fifth one. In this particular scenario, about 10 ms is needed to find each counter-example. These counter-examples show that the proposition $\{Inv$ and not $H\}$ $Iter$ $\{Inv\}$ is not true. They expose a program state satisfying the assertion $\{Inv$ and not $H\}$ such that, after executing the iteration, the resulting program state does not satisfy the invariant anymore. For the first counter-example, the violated part exhibited by the tool is `g <= d`, for the second counter-example, the violated part is `(forall i :  d <= i <= n :  a[i] > x)`. Besides, each time the tool underlines `d := m - 1`. Since the variable `d` is involved in both cases, we have a clue that the error lies in the statement `d := m - 1`.

Finding reasoning errors We can easily imagine that the user forgets to specify the Boolean variable `b` in the loop invariant. The invariant is:

```
Invariant:  unchanged(1, n : a) & unchanged(x)
            & 1 <= g & d <= n + 1 & g <= d
```

```
Data: const n : 0..6 ;
      var x : 0..n ;
      tab a : array [1..n] of 0..n ;

Auxiliary_variables: var g : 0..maxint ;
                     var d : 0..maxint ;
                     var m : 0..maxint ;

Result_variables: var b : boolean ;

Precondition: initialised(x) &
              (forall i: 1 <= i <= n-1 : a[i] <= a[i+1])


Postcondition: unchanged(1, n : a) &
               (b <=> (exist i: 1 <= i <= n : a[i] = x))

Invariant: (unchanged(1, n : a) & unchanged(x)
            & 1 <= g & d <= n + 1 & g <= d
            &&
               (forall i : 1 <= i <= g-1 : a[i] < x)
                   &
               (forall j : d <= j <= n : a[j] > x))
            &
             (b => (exist k : 1 <= k <= n : a[k] = x))

Init: g := 1 ;
      d := n + 1 ;
      b := false

Iter: m := (g + d) div 2 ;
      if (a[m] < x) then
         g := m + 1
      else
         if (a[m] > x) then
             d := m
         else
             b := true
         end
     end

Clot:  skip

Halting_condition: g = d | b = true

Variant: d - g - b
```

Figure 4.5: The binary search algorithm and its specifications

```
            &&
               (forall i : 1 <= i <= g-1 : a[i] < x)
                   &
                (forall j : d <= j <= n : a[j] > x)
```

The system displays the following message in 10 ms:

```
Counter example:  verification of {Inv and H} Clot {Inv} :
Precondition :  a = [1,1], n = 2, x = 2
Invariant and not(H): a = [1,1], b = 1, d = 3, g = 1 , n = 2, x = 2
Postcondition :  a = [1,1], b = 1, d = 3, g = 1 , n = 2, x = 2
```

The user can understand that the example program state proposed before the execution of the closure (`Invariant and not(H)`) is not possible when he thinks at the algorithm operationally: the invariant is too weak. Indeed, nothing in the invariant expresses the fact that b is true if x has been found in a.

Runtime errors Let us assume that we use the condition `g = d + 1` instead of `g = d` in the halting condition of the algorithm. In this case, the system finds input values for which an out of bound array error occurs. The runtime error is underlined in the code:

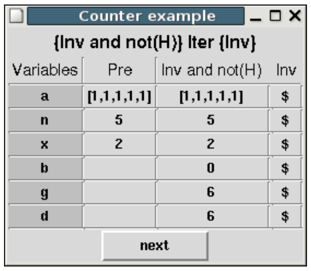
```
if ( a[m]  < x) then ...
```

The first counter-example generated is a borderline case: $n = 0$:

```
Problem in {Inv and not(H)} Iter {Inv} :
Out of bound error :  a[1]
Precondition :  a = [], n = 0, x = 1,
Invariant :  a = [], b = 0, d = 1, g = 1, n = 0, x = 1,
Program point:  a = [], b = 0, d = 1, g = 1, m = 1, n = 0, x = 1
```

The variable state (`Program point`) corresponds to the program state just before the conditional statement for which the condition evaluation involves the runtime error.
Here is another counter-example with `n = 5`:

```
Problem in Inv and not(H) Iter Inv :
Out of bound error :  a[6]
Precondition :  a = [1,1,1,1,1], n = 5, x = 2,
Invariant :  a = [1,1,1,1,1], b = 0, d = 6, g = 6, n = 5, x = 2,
Program point:  a = [1,1,1,1,1], b = 0, d = 6, g = 6, m = 6, n = 5, x
= 2,
```



{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
a	[1,1,1,1,1]	[1,1,1,1,1]	$
n	5	5	$
x	2	2	$
b		0	$
g		6	$
d		6	$

Badly defined assertions We recall that, in an assertion, we cannot have expressions that are not well defined: an out of bound array index and division by zero are prohibited in an assertion as they are prohibited in a statement. With these considerations, assertions may need to be evaluated in a specific order thanks to the operators `&&` and `||` (the formal semantics of these operators is defined in Chapter 5. Suppose now that, in the invariant, we do not specify any order of evaluation between the assertions: we write `&` instead of `&&` between the constraints on the indexes (`g` and `d`) and the quantified assertions ranged on a domain depending on these indexes, as illustrated bellow:

```
...
              1 <= g & d <= n + 1 & g <= d
              &
               (forall i : 1 <= i <= g-1 : a[i] < x)
              &
               (forall j : d <= j <= n : a[j] > x)

...
```

The not well defined expression is underlined:
`(forall i:  1<= i<= g-1 : a[i] < x)`
A counter-example for `n = 5`, is the following:

```
The invariant or the halting condition is not well-defined :
Out of bound error :  a[6]
Precondition:  a = [1,1,1,1,1], n = 5, x = 2,
Invariant:  a = [1,1,1,1,1], b = 1, d = 7, g = 10, i = 6, n = 5, x = 2,
```

Generally, the tool displays counter-examples involving only the program variables state. In this counter-example there appears the bound variable `i`; this shows a value of the bound variable for which an expression in the quantified assertion is not well defined.

The termination of the algorithm In Section 2.1.3, we have explained why, to provide a variant for this algorithm, we need to “convert” the Boolean `b` into an integer value. The tool allows an automatical convertion

of Booleans to integers. Syntactically, the variant is an arithmetic expression that can contain Boolean variables: $true$ and $false$ are resp. converted to 1 and 0. So, the user can write the following variant: `d - g - b`.
Suppose the user writes `g := m` instead of `g := m + 1`. The tool provides the following counter-example:

```
Counter example :  variant does not decrease
Inv:  a = [1,1,1,1,3], b = 0, d = 5, g = 4, n = 5, variant = 1, x = 2,
Inv:  a = [1,1,1,1,3], b = 0, d = 5, g = 4, m = 4, n = 5, variant = 1,
x = 2,
```

If the user chose a negative variant, for example `g - d - b`, the tool prints the following message:

```
Variant not well-defined :  negative expression
Precondition:  a = [1,1,1,1,1], n = 5, variant = 10, x = 1,
Invariant:  a = [1,1,1,1,1], b = 0, d = 6, g = 1, n = 5, variant = 10,
x = 1,
```

4.1.3 The Insert Sort

The insert sort algorithm has been a representative example when we have studied Spark and SMV. This involves a decomposition into subproblems, quantified assertions, a predicate to express the permutation between arrays. To argue the qualities of our tool, let us have a look on the way our tool manages the verification of this algorithm. Importantly, subproblems are proved completely independently from each other. Thus, checking the main problem does not require the code of the subproblem, it relies on its specification instead. Besides, it is allowed that a subproblem is only specified (declarations, precondition and postcondition). Notice that our tool does not consider parameters, but only involves global variables.

Figure 4.6 depicts the two subproblems composing the insert sort algorithm. Concretely each subproblem is identified by its file and we can go easily from one to the other. In both, we use a predicate `permut(a , a_0 , 1 , i , 1 , i)` expressing that the subarray $a[1..i]$ is a permutation of the initial subarray $a_0[1..i]$. a and a_0 denote functions from the index set to the set of values: a_0 stands for this function at the initial state. The predicate `unchanged(i + 1 , n :a)` expresses that the subarray $a[i+1..n]$ is unchanged relating to the array values specified by a_0. The variables involved in pre and postconditions of the subproblem are variables of the main problem.

Proving the algorithm correctness The subproblems are proved independently from each others. With `n :  0..4`, the subproblem `insertsortSP`

```
Data: const n: 0..4;
      tab a : array [1..n] of 1..n;

Auxiliary variables: var i : 0..maxint;

Precondition: initialised(a)

Postcondition: (forall k: 1 <= k <= n-1 :  a[k] <= a[k+1]))
               & permut(a , a_0 , 1 , n , 1 , n)

Invariant: 0 <= i <= n &&
           (forall k: 1 <= k <= i-1 : a[k] <= a[k+1])
           & permut(a , a_0 , 1 , i , 1 , i)
           & unchanged(i + 1 , n : a)

Init: i := 0

Iter: sp(triinsertSP.in) ; i := i + 1

Clot: skip

Halting_condition: i = n
```

Figure 4.6: The main problem of the insertion sort algorithm

is proved in 240 ms; the main problem is proved in 650 ms. Notice that the parametrisation `n :0..4` takes the borderline cases into account : it must also be correct for an empty array and for an array with one element.

A violated precondition If the user considers that the subproblem inserting `a[i]` at the right place in `a[1..i]` requires that `i > 0` [1], the tool gives the following counter-example in the main problem feedback area:

```
Problem in Inv and not(H) Iter Inv :
The precondition of the subproblem triinsertSP is not satified
Precondition :  a = [1..4,1..4,1..4,1..4], n = 4,
Invariant :  a = [1..4,1..4,1..4,1..4], i = 0 , n = 4,
Program point:  a = [1..4,1..4,1..4,1..4], i = 0 , n = 4,
```

In this example, the elements of `a` are not determined; `1..4` is the domain given at the array declaration; in fact, no matter the values of `a`, the subproblem precondition is violated when `i = 0`.
In the subproblem editor, the violated part of the precondition is underlined:
`Precondition: 0 < i  & i < n && ...`

[1] This can be justified since inserting `a[1]` in `a[1]` is not useful

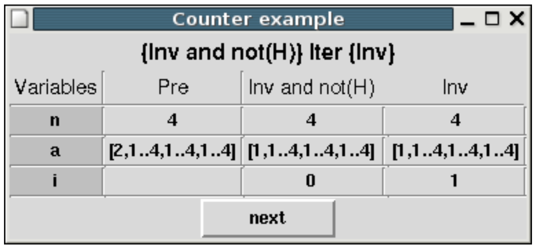
```
Data: const n : 0..4;
      var i : 1..n;
      tab a : array [1..n] of 1..n;

Auxiliary variables: var j : 0..maxint;
                     var x : 0..maxint;

Precondition: 0 <= i & i < n &&
              (forall j : 1 <= j <= i-1 : a[j] <= a[j+1]))

Postcondition: unchanged(i) &&
               unchanged(i+2 , n : a) &
               (forall k : 1 <= k <= i : a[k] <= a[k+1]) &
               permut(a , a_0 , 1 , i+1 , 1 , i+1)

Invariant: unchanged(i) &
           (1 <= j & j <= i+1)
           && (unchanged(i+2 , n : a) & x = a_0[i+1] &
           (forall k : j <= k <= i: a_0[k] = a[k+1]) &
           unchanged(1 , j : a) &
           (forall k :j+1 <= k <= i+1 : a[k]>x)

Init: j := i+1 ; x := a[j]

Iter: a[j] := a[j-1] ; j := j-1

Clot: a[j] := x

Halting_condition: j = 1 || a[j-1] <= x
```

Figure 4.7: `triinsertSP.in`: the subproblem of the insertion sort algorithm

A too weak invariant Assume that we keep the same specification and the same code, but that we forget to mention, in the loop invariant, that the subarray `a[i+1..n]` is unchanged. The invariant is:

```
0 <= i <= n &&
  (forall j : 1 <= k < i: a[k] <= a[k+1]))
  & permut(a , a_0 , 1 , i , 1 , i)
```

Assuming `n = 4`, our system gives the following counter-example in 10 ms.



Counter example

{Inv and not(H)} Iter {Inv}

Variables	Pre	Inv and not(H)	Inv
n	4	4	4
a	[2,1..4,1..4,1..4]	[1,1..4,1..4,1..4]	[1,1..4,1..4,1..4]
i		0	1

The variable state satisfying the given invariant shows that $a[1]$ is different from $a_0[1]$, which is strange because if $i = 0$, nothing in the array can have been permuted. The user should discover that its invariant is not strong enough: all the values that we do not permute must stay unchanged.

Runtime error If we use a strict evaluation in the closure of the subproblem: `j = 1 | a[j-1] <= x`. The tool discovers a runtime error and the involved expression is underlined:

```
   Halting_condition:  j = 1 | a[j-1] <= x
```

A counter-example is the following:

```
The invariant or the halting condition is not well defined :
Out of bound error :  a[0]
Precondition :  a = [1,0,0..4,0..4] , i = 1 , n = 4 ,
Invariant :  a = [1,1,0..4,0..4] , i = 1 , j = 1 , n = 4 , x = 0 ,
```

We see that the invariant considers a program state where $j = 1$, the evaluation of the halting condition in this case, generates a runtime error since the two parts of the disjunction are evaluated simultaneously.

4.1.4 The Next Permutation

The automatic proof of the correctness of the next permutation algorithm is a real challenge: this algorithm requires a assertion language with a lot of expressivity, and it needs a technical way adapted to make so tedious proofs. We have already submitted this algorithm to a manual proof, and to automatic (or partially automatic) proofs using theorem proving. Let us have a look on the way our tool will manage this algorithm. Below, we can first observe the subproblems specifications exactly as they are written in our tool.

SP1: The largest index such that $a[i] < a[i+1]$.

```
Precondition: initialised(1,n :a) & n > 0
Postcondition: unchanged(1,n : a) &
              (0 <= i & i < n) &&
              (forall k : 1 <= k <= n-1 : a[k] >= a[k+1])&
              (i = 0 &  ||  a[i] < a[i+1])
```

SP2: $a[j]$ is the last element greater than $a[i]$, on its right.

```
Precondition: 0 < i & i < n &&
              (forall k1: i+1 <= k1 <= n-1: a[k1] >= a[k1+1])
              & a[i] < a[i+1]
Postcondition: unchanged(1,n: a) & unchanged(i) &
               (i < j & j <= n) &&
               (j = n && a[j] > a[i]) |
               (j < n && (a[j+1] < a[i] & a[j] > a[i]))
```

SP3: The reversing of the subarray $a[i+1..n]$.

```
Precondition: initialised(1,n : a)  & 0 < i  & i < n
Postcondition: unchanged(i) && unchanged(1 , i: a)
               & (forall k : i + 1 <= k <= n :
                              a[k] = a_0[n-(k-(i+1))])
```

The formalisation is very similar to the formalisation expressed in Section 2.1.5. Let us have a look on the specification of the main algorithm on Figure 4.8. As seen in Section 2.1.5, this specification needs high expressivity: we can use quantifiers on arrays, a generalised quantifier for a number of occurrences `#`, relations between arrays: `<<` , `>>` and `>>=` for the lexicographic order. To express that `a` is a permutation of `1..n`, we say that each value from `1..n` must have one and only one occurrence in `a`. However, the algorithm is a simple sequence of subproblem calls.

Proving the algorithm correctness For an array of size `n:0..5`, it takes 2.18 sec. To prove the correctness of the subproblems, assuming $n : 0..5$, it takes 80 ms for SP1, 200 ms for SP2 and 250 ms for SP3.

A too weak specification of the subproblem SP2 Let us now assume that we try to prove the correctness of the main algorithm with an inadequate (i.e., too weak) specification of a subproblem. More specifically, we forget the constraint $a[j] > a[i]$ in the postcondition of SP2. For $n = 5$, we get the following counter-example for the main algorithm.

```
Counter example :  verification of {Pre} Instr {Post}
Pre :   a = [3,5,4,2,1] , n = 5 ,
Post :   a = [2,1,3,4,5] , b = 1 , i = 1 , j = 4 , n = 5 , temp = 2 ,
```

The violated part is `a_0 << a`.

```
Data: const n = 5;
      tab a : array [1..n] of 1..n;

Auxiliary_variables: var b: boolean;
                     var i: 0..maxint;
                     var j: 0..maxint;
                     var temp: 0..maxint;

Precondition: (forall k : 1 <= k <= n :
                  (# l:1 <= l <= n : a[l] = k) = 1)

Postcondition:
(b =>
  (
   ((forall k:1 <= k <= n : (# l:1 <= l <= n : a[l] = k) =1)
      &
      a_0 << a)
    &&
   (forall x[1..n] : 1 <= x <= n :
               ((forall k:1 <= k <= n :
                  (# l:1 <= l <= n : x[l] = k) = 1) & x >> a_0)
                => x >>= a
   )
))
&
((! b) =>
   (unchanged(1,n:a) & (forall k : 1 <= k <= n-1: a[k] > a[k+1])))

Instr:
     sp(sp1.in);
     if(i<>0) then
         b:= true;
         sp(sp2.in);
         temp := a[j];
         a[j] := a[i];
         a[i] := temp;
         sp(sp3.in)
      else
          b:= false
      end
```

Figure 4.8: The main algorithm of the next permutation, together with its specification

A too strong postcondition of the subproblem SP2 On the other hand, if we forget the special case ($j = n$ && $a[j] > a[i]$) in the postcondition of SP2, we get a too strong postcondition (i.e., no terminating algorithm may implement such a specification). This error, which was actually made unintentionally when we first solved this problem, cannot logically be detected in the proof of the main algorithm. However, it is detected if we analyse the subproblem itself: if we attempt to construct the algorithm according to this wrong specification, we get an out of bound error prediction; if we check a correct algorithm with respect to the wrong specification, we get counter-examples for $\{Inv$ and $H\}$ $Clot$ $\{Post\}$.

4.2 Discussion

In this chapter, we have presented an automatic tool that we have completely elaborated and that insures complete and sound verification. If the user rigorously follows the structured programming method (which is a condition that corresponds to our pedagogical objectives), the tool is able to prove the correctness or to find counter-examples. The user cannot get false warnings and there are no forgotten errors; so, the user can be confident in the feedback; he can try to understand his reasoning errors without doubting of the error.

The example of the Indian exponentiation has shown the ability of our tool to manage non linear expressions such as multiplication and exponentiation between variables. The tool easily detects reasoning errors and overflow; the user does not have to write any specific rules, nor to help the verification by making an interactive proof. With the binary search algorithm, we have introduced verification scenarii involving quantifiers. We have seen that the verification result is precise, no matter how the assertions are formalised, whereas ESC/Java2 requires the user to help the verification by writing assertions in a way instead of another. The insertion sort algorithm also shows that manipulating quantifiers does not involve any problem of completeness or soundness; it also uses a predicate to express that a subarray is a permutation of another subarray. This algorithm is an example of decomposition into subproblems and the tool follows the principles of the structured programming method. Again the verification feedback is precise for both subproblems; notice that the subproblem that inserts an element at the right place in a subarray cannot be verified by ESC/Java2, nor by Spark. The example of the next permutation shows that no matter the complexity of the proof of the correctness, the tool guarantees a complete and sound verification. This example uses a specific quantifier to express the number of occurrences and specific relation operators to express the lexicographic order. A quantifier on array is used and does not involve any problem to give a precise feedback about the correctness of the algorithm. Nowhere in these exercises, the student needs an additional knowledge beyond the structured programming method.

Our tool has all these advantages because we have restricted our ambition to our pedagogical context: we do not need to cover a large and complex programming language like Java: we consider that, for teaching algorithmics, we do not need a too much elaborated language. The simpler it is, the better it is: our language just needs to consider the basic notions introduced in Section 2.1.1. The language considered is imperative, it is an approximate subset of Pascal; it is simple in that there are no parameters, no local declarations, and each algorithm must have a specific format. Calls to subproblems are permitted; their variables must be global to the problem. We use fixed size arrays, integers (limited to finite domains) and Booleans. On the opposite, one can observe that the specification language is expressive enough to be really usable: we express relations between values and relations between arrays, we use existential and universal quantifiers on variables on finite domains, other generalised quantifiers such as sum, maxima, number of occurrences, quantifiers on arrays, and some predefined predicates are useful as that of the permutation. To refer to the initial value of a variable `x`, we write `x_0`.

This context allows us to follow an approach which is rather original: we translate the program together with its related assertions into a set of constraints. Then, constraints programming techniques can be used to prove that the program is correct or to exhibit errors. Completeness and soundness stems from the fact that we exhaustively solve the constraints (unless counter-examples are produced and the user decides to give up their generation). Of course, this is possible only because we make the variables range over small domains. We believe that this limitation is not harmful, in our context.

Part II

Definition and Implementation of the Languages

Chapter 5

Definition of the Languages

In the previous chapter, we have given the reader a general idea of the approach used by the tool, by demonstrating our tool abilities. The programming language of MPVS is imperative; it can be viewed as a small subset of Pascal: there are no parameters, no local declarations and the form of the algorithm is restricted. On the opposite, the specification language is very expressive. In this chapter, after fixing some mathematical notations, we formally define the programming language and the assertion language accepted by our tool. The goal of this chapter is to provide a complete information about the languages so that the user can fully understand the specific behaviour of the tool.

5.1 Mathematical Notations

This section can be skipped at first reading. It gives notations to be understood for a deeper reading of the technical point of view in this Chapter.

5.1.1 Partial functions

We need to define several ways to update a function. Let M and N be two sets. A partial function from M to N is denoted by $f : M \nrightarrow N$. The domain of f is noted $dom(f)$ and the codomain of f, $codom(f)$.

- ϕ is the empty partial function, undefined for all $m \in M$.
- Let $m \in M, n \in N$.

  $f[m/n]$ is the function $f' : M \nrightarrow N$ that updates f such that:

  forall $i \in dom(f) \cup \{m\}$:

  - $f'(i) = f(i)$ if $i \neq m$
  - $f'(m) = n$

- Let $m \in M, n \in N$.
  - If $m \notin dom(f)$, $f[m \mapsto n]$ is the function $f' : M \nrightarrow N$ that updates f such that:
    for all $i \in dom(f) \cup \{m\}$:
    - $f'(i) = f(i)$ if $i \neq m$
    - $f'(m) = n$
  - If $m \in dom(f)$ and $f(m) = n$ then $f[m \mapsto n] = f$.
  - If $m \in dom(f)$ and $f(m) \neq n$ then $f[m \mapsto n]$ is not well-defined and generates an *error*.

- Let $m \in M$, $n \in N$ and $n' \in \mathcal{P}(N)$.
  - If $m \notin dom(f)$, $f[m \hookrightarrow n']$ is the function $f' : M \nrightarrow N$ that updates f such that:
    for all $i \in dom(f) \cup \{m\}$:
    - $f'(i) = f(i)$ if $i \neq m$
    - $f'(m) = choice(n')$
      where
      $$\begin{array}{lll} choice: & \mathcal{P}(N) & \rightarrow N \\ & n' & \rightarrow n \end{array}$$
      such that $n \in n'$.
  - If $m \in dom(f)$ and $f(m) \in n'$ then $f[m \hookrightarrow n'] = f$.
  - If $m \in dom(f)$ and $f(m) \notin n'$ then $f[m \hookrightarrow n']$ is not well-defined and generates an *error*.

5.2 A Simple Programming Language

5.2.1 Abstract Syntax

We first present in Table 5.1, the allowed *phrase types* of our language, and we specify in Table 5.2, the language *syntax*. Types admitted are integer and Boolean. Programs can use integer constants, Boolean and integer variables, and arrays of integers; all these objects are defined on positive finite domains which are defined by expressions that may depend on constants. A program is a list of subproblems such that each subproblem called must be listed after the subproblem calling it. Each subproblem is composed of a list of declarations and a body. The body can be a simple sequence of statements, a conditional statement or a loop statement.

$decl \in Decl$	declarations
$body \in Body$	subproblem cores
$com \in Com$	statements
$aexpr \in Aexpr$	numerical expressions
$bexpr \in Bexpr$	Boolean expressions
$spr \in SPr$	subproblems
$be \in Be$	numerical expressions for bounded domains
$l \in L$	numerical expressions for array length
$bop \in Bop$	strict Boolean operators
$bopltr \in BopLtr$	lazy Boolean operators
$bnot \in Bnot$	unary Boolean operator
$cop \in Cop$	relation operators
$eop \in Eop$	equality operators
$aop \in Aop$	numerical operators
$i \in N$	integers
$bool \in Bool$	Booleans
$c \in C$	constant names
$x \in Id$	integer variable names
$b \in B$	Boolean variable names
$tab \in Tab$	array names
$p \in P$	subproblem names

Table 5.1: Syntax Domains, Generic Variables

5.2.2 Operational Semantics

In this section, we formally describe the execution of programs written in this language, by means of an operational semantics. The first part of our definition is the definition of the semantic sets. The choice of these sets is closely related to our implementation.

5.2.2.1 Constants and semantic values

The constant **maxint** is the greatest integer representable in our tool; its value is implementation dependent.

The domain **Val** is the finite natural domain [0..maxint]. It represents the set of values that we can assign to our program variables.

The set D is the set of all intervals for which the values are included in Val; $D \subseteq \mathcal{P}(Val)$.

$$D = \{[binf..bsup] \mid binf \in Val, bsup \in Val\}$$

Declarations

decl ::= *const c* : [*be..be*]
| *tab*[1..*l*] of [*be..be*]
| *x* : [*be..be*]
| *b* : *Boolean*
| *decl**

l ::= *i* | *c* | *l aop l*

be ::= *l*| **maxint**

Statements

com ::= skip
| *x* **:=** *aexpr*
| *b* **:=** *bexpr*
| *tab*[*aexpr*] **:=** *aexpr*
| **if** *bexpr* **then** *com* **else** *com*
| *p*
| *com com**

Expressions

aexpr ::= *c*
| *i*
| *x*
| *tab*[*aexpr*]
| *aexpr aop aexpr*

bexpr ::= *bool*
| *b*
| *aexpr cop aexpr*
| *bexpr bop bexpr*
| *bexpr bopltr bexpr*
| *bnot bexpr*

Operators

bop ::=| | & | *eop*

bopltr ::=|| | &&

bnot ::=!

aop ::=+ | − | ∗ | *div* | *mod*

cop ::=< | > | <= | >= | *eop*

eop ::== | <>

Subproblems and Programs

body ::= *com*
| *com com com bexpr*

spr ::= *decl body*

prog ::= *p* : *spr* (*p* : *spr*)*

Table 5.2: Language Syntax

The abstract value **neg** represents any strictly negative value.

Generally speaking, our language does not handle negative numbers. Nevertheless, there are few situations where the user has to specify empty intervals by means of negative upperbounds. Thus, we accept a limited use of negative expressions.

5.2.2.2 The static environment

As we have explained in Chapter 4, one of the characteristics of MPVS is that we do not have to fix the arrays size, which may be useful to automatically verify borderline cases involving arrays. To allow it, we can choose a finite domain for the constant defining the array size. Also, it may be useful that for each variable, we can choose a domain related to the constants: For example, in the next permutation algorithm, the data domain must be dependent on the array size: `tab a:  array[1..n] of [1..n]`. The role of the static environment is to fix the array sizes and the variables domains according to fixed values for each constant.

The *static environment* is represented in Figure 5.1 and specifies:

$$Td \ni td : C + Id + Tab \nrightarrow Val + D + (Val \times D)$$

For readability reasons, we may mention it as a triple function $(td_c, td_{id}, td_{tab})$ defined as follows:

$$\begin{array}{lll} td_c & : C & \nrightarrow Val \\ td_{id} & : Id & \nrightarrow D \\ td_{tab} & : Tab & \nrightarrow Val \times D \end{array}$$

Figure 5.1: The static environment

- a value for each constant,
- a domain of values for each integer variable,
- a fixed size for each array plus a domain of values for its elements.

5.2.2.3 The environment

According to one specific static environment *td*, an *environment e* associates each simple variable of the program to a value or to *noval* if the variable is not initialised, and each array to a partial function that links the indexes to a value (or *noval*). We call Env_{td} the set of the environments compatible with the static environment *td*: Figure 5.2 expresses that

- for constants the value must be the one specified in the static environment *td*,
- for initialised Boolean variables, the value must be $true$ or $false$,
- for initialised integer variables, the value must be in the domain specified in *td*,
- for arrays, let f be the function associated to the identifier: $dom(f)$ is the set of indexes of the array, i.e., an interval from 1 to the array length (specified in *td*), and $codom(f)$ is the set of elements of the array; they must be included in the domain specified in *td*.

5.2.2.4 The interpretation set

According to a static environment *td*, the semantics of a subproblem is an environment-transforming function whose domain is the set of environments $e_{in} \in Env_{td}$. So, for each subproblem, we associate a function

$$f = Env_{td} \rightarrow Env_{td} + \{error, \perp\}$$

$Env_{td} \ni e :$

$$C{+}Id{+}B{+}Tab \nrightarrow Val{+}(N \nrightarrow Val{+}\{noval\}){+}\{true, false\}{+}\{noval\}$$

For readability reasons, we may mention it as a quadruple function $(e_c, e_{id}, e_b, e_{tab})$ defined as follows:

$$\begin{array}{lll} e_c & : C & \nrightarrow Val \\ e_{id} & : Id & \nrightarrow Val + \{noval\} \\ e_b & : B & \nrightarrow \{true, false, noval\} \\ e_{tab} & : Tab & \nrightarrow (N \nrightarrow Val + \{noval\}) \end{array}$$

such that
for all $c \in dom(e_c) : e_c(c) = td_c(c)$
for all $id \in dom(e_{id}) : e_{id}(x) \in td_{id}(x)$ or $e_{id}(x) = noval$
for all $tab \in dom(e_{tab})$:
  let $(l, d) = td_{tab}(tab)$:
  $dom(e_{tab}(tab)) = [1..l]$ and for all $k \in dom(e_{tab}(tab))$: $e_{tab}(tab)(k) \in d$ or $e_{tab}(tab)(k) = noval$

Figure 5.2: The environment

. For each $e_{in} \in Env_{td}$, the execution of the subproblem

- may not terminate by generating a runtime error: $f(e_{in}) = error$;
- may not terminate: $f(e_{in}) = \perp$;
- may terminate with the program environment e_{out}: $f(e_{in}) = e_{out}$.

We call the *interpretation set*, $itp \in Itp$, the function that associates each subproblem identifier to its semantics.

$$itp : P \nrightarrow \{Env_{td} \rightarrow Env_{td} + \{error + \perp\}\}$$

The semantics of a program is a partial function *itp* where *dom*(*itp*) is the set of the identifiers of all subproblems appearing in the definition of the program given by $prog ::= p : spr\ (p : spr)^*$.

5.2.2.5 Type Checking analysis

We consider that expressions between double brackets are well defined: each involved identifier exists and belongs to an appropriate syntactic set.

5.2.2.6 Declarations

The semantics of a program is based on a static environment *td* which is defined from program declarations. To give an intuitive idea of how *td* is initialised, let us first look at an example.

Example:

```
(1) const n : 1..6 ;
(2) const m : 0..n + 1 ;
(3) var x : 0..m ;
(4) tab a :array[1..n] of 1..m ;
```

(1) An arbitrary value for `n` included in `1..6` is chosen, say `n = 3`; **(2)** to choose a value for `m` among the domain `0..n + 1`, the value of `n` must have been chosen; in the example, we can choose for instance `m = 4`. **(3)** The `x` domain is fixed to `0..4`. **(4)** The length of `a` is fixed to 3 and the domain of its elements is fixed to `1..4`.

$$\mathcal{D} : Decl \nrightarrow Td \rightarrow Td + \{error\}$$

$$\begin{array}{lcl}
\mathcal{D}[\![c : [be_1..be_2]]\!]\ td & = & td[c \hookrightarrow [i_1..i_2]] \\
 & & \text{where } i_1 = \mathcal{E}val_c\ [\![be_1]\!]\ td \\
 & & \text{and } i_2 = \mathcal{E}val_c\ [\![be_2]\!]\ td \\
\mathcal{D}[\![x : [be_1..be_2]]\!]\ td & = & td[x \mapsto [i_1..i_2]] \\
 & & \text{where } i_1 = \mathcal{E}val_c\ [\![be_1]\!]\ td \\
 & & \text{and } i_2 = \mathcal{E}val_c\ [\![be_2]\!]\ td \\
\mathcal{D}[\![b : Boolean]\!]\ td & = & td \\
\mathcal{D}[\![tab[1..l]\ of\ [be_1..be_2]]\!]\ td & = & td[tab \mapsto (i_1, [i_2..i_3])] \\
 & & \text{where } i_1 = \mathcal{E}val_c\ [\![l]\!]\ td \\
 & & \text{and } i_2 = \mathcal{E}val_c\ [\![be_1]\!]\ td \\
 & & \text{and } i_3 = \mathcal{E}val_c\ [\![be_2]\!]\ td \\
\mathcal{D}[\![\ ]\!]\ td & = & td \\
\mathcal{D}[\![decl_1\ decl]\!]\ td & = & \mathcal{D}[decl]\ (\mathcal{D}[decl_1]\ td)
\end{array}$$

where $\mathcal{E}val_c$ is a function that evaluates arithmetic expressions only involving constant values that must be determined in the static environment.

$$\mathcal{E}val_c : Be \rightarrow Td \rightarrow Val + \{error\}$$

$$\begin{array}{lcl}
\mathcal{E}val_c[\![c]\!]\ td & = & td_c(c) \text{ if } c \in dom(td_c) \\
 & & error \text{ otherwise} \\
\mathcal{E}val_c[\![i]\!]\ td & = & i \\
\mathcal{E}val_c[\![be_1\ aop\ be_2]\!]\ td & = & v \text{ if } v \in Val \\
 & & error \text{ otherwise}
\end{array}$$

where $v = \mathcal{A}op_{strict}[\![aop]\!]\ (\mathcal{E}val_c[\![be_1]\!]\ td)\ (\mathcal{E}val_c[\![be_2]\!]\ td)$
The function $\mathcal{A}op_{strict}$ is defined in Table 5.4

The semantics of a declaration *decl* modifies the static environment *td*, unless the identifier declared by *decl* is already in *dom*(*td*); it this case, the static information given by *decl* must match the current static environment *td*.

The static environment td for a program is defined from the declarations of all its subproblems.

$$\mathcal{D}prog : Prog \to Td + \{error\}$$

Let $spr = decl\ body$

$$\begin{array}{lcl}
\mathcal{D}prog[\![p : spr]\!] & = & \mathcal{D}[\![decl]\!]\ \phi \\
\mathcal{D}prog[\![p : spr\ (p_i : spr_i)^+]\!] & = & \mathcal{D}[\![decl]\!]\ td \\
 & & \text{where } td = \mathcal{D}prog[\![(p_i : spr_i)^+]\!]\ \phi \\
 & & error \text{ if } \mathcal{D}prog[\![(p_i : spr_i)^+]\!]\ \phi = error
\end{array}$$

The static environment td is updated with the union of the declarations of the subproblems list. This formal definition expresses that there cannot be coherence problems in the static environment such as, for example, a same variable involved in two subproblems with two different domains of values is not allowed.

5.2.2.7 Expressions

The semantics of an expression is the result of its evaluation. The environment is not modified, the evaluation function just consults it. The semantics is described as follows:

Numerical expressions Usually, the evaluation of a numerical expression results into a numerical value or possibly an error because, for example, of an array out-of-bound error, of a division by zero or simply because the variable to be evaluated is not initialised.

In general, MPVS does not handle negative values. To cover some cases that may become more convenient, we use abstract domains of values and abstract interpretation techniques: if we have enough information to testify that the result is negative, the evaluation result is an "abstract" value, *neg*. If we do not have enough information to testify that the result is negative or has a precise numerical value, the result of the evaluation is *error*. Let us detail our specific evaluation function:

$$\mathcal{E}val : Aexpr \to Env_{td} \to Val + \{error, neg\}$$

$$\begin{array}{lcl}
\mathcal{E}val[\![c]\!]\ e & = & e_c(c) \\
\mathcal{E}val[\![i]\!]\ e & = & i \\
\mathcal{E}val[\![x]\!]\ e & = & e_{id}(x) \text{ if } e_{id}(x) \neq noval \\
 & & error \text{ otherwise} \\
\mathcal{E}val[\![tab[aexpr]]\!]\ e & = & e_{tab}(tab)(\mathcal{E}val[\![aexpr]\!]\ e) \\
 & & \text{if } \mathcal{E}val[\![aexpr]\!]e \in dom(e_{tab}(tab)) \\
 & & error \text{ otherwise} \\
\mathcal{E}val[\![aexpr_1\ aop\ aexpr_2]\!]\ e & = & \mathcal{A}op(aop)\ (\mathcal{E}val[\![aexpr_1]\!]\ e)\ (\mathcal{E}val[\![aexpr_2]\!]\ e)
\end{array}$$

where $\mathcal{A}op$ describes the semantics of the arithmetic operators and is defined below:

Arithmetic operators Our domain of values is not the usual domain of integers. In fact, $Val + \{Neg\}$ is an abstract domain such that

$$\begin{array}{rcccl} abs: & Z & \rightarrow & Val + \{neg\} & \\ & i & \rightarrow & i & \text{if } i \geq 0 \\ & & \rightarrow & neg & \text{if } i < 0 \end{array}$$

[1] Usually, in abstract interpretation, we add in the abstract domain, the value $\top$ to get an order relation $\preceq$ on the precision on the approximations of the abstract values. In our context, we should have $neg \preceq \top$ and $i \preceq \top$ where $i \in Val$. However, the value $\top$ will be considered as an error because we cannot handle this case of expression evaluation in our tool.

So, the semantics of the "abstract" operators $aop \in Aop$ is defined as follows:

$$\mathcal{A}op : Aop \rightarrow Val+\{error, neg\} \rightarrow Val+\{error, neg\} \rightarrow Val+\{error, neg\}$$

Table 5.3 expresses error propagation : an operation involving an error generates an error.

If we do not consider the error propagation, we define the following function

$$\mathcal{A}op_A : Aop \rightarrow Val + \{neg\} \rightarrow Val + \{neg\} \rightarrow Val + \{neg\} + \{error\}$$

Let $v_1, v_2 \in Z$, the function $\mathcal{A}op_A$ has the following property:

$$\begin{array}{lcll} Aop_A[\![aop]\!]\ abs(v_1)\ abs(v_2) & = & abs(v_1\ aop_Z\ v_2) & \text{if } v_1\ aop_Z\ v_2 \neq error \\ & & error & \text{otherwise} \end{array}$$

if for all $x \in abs(v_1)$, for all $y \in abs(v_2)$: $abs(x\ aop\ y) = abs(v_1\ aop_Z\ v_2)$ where aop_Z is the arithmetic operator usually manipulated between integers (the concrete operator corresponding to aop). For the other cases, the result abstraction is not manageable, the result is $error$. Notice that an $error$ can result from something else than an abstract problem: a badly defined arithmetic expression such as a division by zero, a modulo by zero or a modulo with negative values. The complete definition is given in Table 5.6.

To make this definition, we define the semantics of the operations between values from Val.

$$\mathcal{A}op_{strict} : Aop \rightarrow Val \rightarrow Val \rightarrow Val + \{error, neg\}$$

[1]To be nicer, we mention Z for the usual integers domain, in fact, our usual integers domain is $[-\infty, \mathbf{maxint}]$.

The behaviour of the abstract operators in this context is very similar to the behaviour of the corresponding operators between usual integers. We may just have a loss of information with the minus operator whose evaluation may result to a negative value. Table 5.4 gives details of this function.

We also define in Table 5.5 the semantics of the operators between negative values.

$$\mathcal{A}op_{neg} : Aop \to \{neg\} \to \{neg\} \to Val + \{error, neg\}$$

For most operators, the result is not manageable. A positive result will never be precise enough.

Boolean expressions The evaluation of a Boolean expression should result to a Boolean value, i.e., $true$ or $false$. As for numerical expressions, the evaluation may generate an error, because of an error in a numerical subexpression, or because the Boolean variable that we want to evaluate is not initialised.

$$\mathcal{B} : Bexpr \to Env_{td} \to \{true, false, error\}$$

$$\begin{array}{lcl}
\mathcal{B}[\![bool]\!]\ e & = & bool \\
\mathcal{B}[\![b]\!]\ e & = & e_b(b) \text{ if } e_b(b) \neq noval \\
 & & error \text{ otherwise} \\
\mathcal{B}[\![aexpr_1\ cop\ aexpr_1]\!]\ e & = & \mathcal{C}op[\![cop]\!]\ (\mathcal{E}val[\![aexpr_1]\!]\ e)\ (\mathcal{E}val[\![aexpr_2]\!]\ e) \\
\mathcal{B}[\![bexpr_1\ bop\ bexpr_2]\!]\ e & = & \mathcal{B}op[\![bop]\!]\ (\mathcal{B}[\![bexpr_1]\!]\ e)\ (\mathcal{B}[\![bexpr_2]\!]\ e) \\
\mathcal{B}[\![bexpr_1\ bopltr\ bexpr_2]\!]\ e & = & \mathcal{B}op[\![bopltr]\!]\ (\mathcal{B}[\![bexpr_1]\!]\ e)\ (\mathcal{B}[\![bexpr_2]\!]\ e) \\
\mathcal{B}[\![bnot\ bexpr]\!]\ e & = & true \text{ if } \mathcal{B}[\![bexpr]\!]\ e = false \\
 & & false \text{ if } \mathcal{B}[\![bexpr]\!]\ e = true \\
 & & error \text{ otherwise}
\end{array}$$

Relational operators

$$\mathcal{C}op : Cop \to Val{+}\{neg, error\} \to Val{+}\{neg, error\} \to \{true, false, error\}$$

Table 5.8 expresses the error propagation : a comparison involving an error generates an error. For comparisons that do not have any error term, we define the following function:

$$\mathcal{C}op_A : Cop \to Val + \{neg\} \to Val + \{neg\} \to \{true, false\} + \{error\}$$

The behaviour of the abstract relational operators is similar to the behaviour of the corresponding operators between usual integers. We will just have a loss of information if we compare two negative values. In the other cases, we have enough information to get a precise comparison result. As it is described in Table 5.9, if we compare a strictly negative value with a value i from Val ($i \geq 0$), the result is clearly defined; if we compare two values from

$\mathcal{A}op : Aop \rightarrow Val + \{error, neg\} \rightarrow Val + \{error, neg\} \rightarrow Val + \{error, neg\}$

aop a_1 a_2	$error$	$Val + \{neg\}$
$error$	$error$	$error$
$Val + \{neg\}$	$error$	$\mathcal{A}op_A[\![aop]\!]\ a_1\ a_2$

where Aop_A is defined in Table 5.6.

Table 5.3: Error propagation in an arithmetic expression evaluation

$$\mathcal{A}op_{strict} : Aop \rightarrow Val \rightarrow Val \rightarrow Val + \{error, neg\}$$

$$\begin{array}{lcll}
\mathcal{A}op_{strict}[\![+]\!]\ i_1\ i_2 & = & i_1 + i_2 & \text{if } i_1 + i_2 \in Val, \\
 & & error & \text{otherwise} \\
\mathcal{A}op_{strict}[\![-]\!]\ i_1\ i_2 & = & i_1 - i_2 & \text{if } i_1 \geq i_2, \\
 & & neg & \text{if } i_2 > i_1, \\
 & & error & \text{otherwise} \\
\mathcal{A}op_{strict}[\![*]\!]\ i_1\ i_2 & = & i_1 * i_2 & \text{if } i_1 * i_2 \in Val, \\
 & & error & \text{otherwise} \\
\mathcal{A}op_{strict}[\![div]\!]\ i_1\ i_2 & = & i_1\ div\ i_2 & \text{if } i_2 \neq 0, \\
 & & error & \text{otherwise} \\
\mathcal{A}op_{strict}[\![mod]\!]\ i_1\ i_2 & = & i_1\ mod\ i_2 & \text{if } i_2 \neq 0, \\
 & & error & \text{otherwise}
\end{array}$$

Table 5.4: Evaluation of arithmetic expressions between values $v \in Val$

$$\mathcal{A}op_{neg} : Aop \rightarrow \{neg\} \rightarrow \{neg\} \rightarrow Val + \{error, neg\}$$

$$\begin{array}{lcl}
\mathcal{A}op_{neg}[\![+]\!]\ a_1\ a_2 & = & neg \\
\mathcal{A}op_{neg}[\![-]\!]\ a_1\ a_2 & = & error \\
\mathcal{A}op_{neg}[\![*]\!]\ a_1\ a_2 & = & error \\
\mathcal{A}op_{neg}[\![div]\!]\ a_1\ a_2 & = & error \\
\mathcal{A}op_{neg}[\![mod]\!]\ a_1\ a_2 & = & error
\end{array}$$

Table 5.5: Evaluation of arithmetic expressions between strictly negative values

$$\mathcal{A}op_A : Aop : Val + \{neg\} \to Val + \{neg\} \to Val + \{error, neg\}$$

$+$ a_1 / a_2	neg	Val
neg	$\mathcal{A}op_{neg}[\![+]\!]\ a_1\ a_2$	$error$
Val	$error$	$\mathcal{A}op_{strict}[\![+]\!]\ a_1\ a_2$

$-$ a_1 / a_2	neg	Val
neg	$\mathcal{A}op_{neg}[\![-]\!]\ a_1\ a_2$	$error$
Val	neg	$\mathcal{A}op_{strict}[\![-]\!]\ a_1\ a_2$

div a_1 / a_2	neg	Val
neg	$\mathcal{A}op_{neg}[\![div]\!]\ a_1\ a_2$	$\begin{cases} 0 \\ \text{if } a_1 = 0 \\ neg \\ \text{otherwise} \end{cases}$
Val	$\begin{cases} error \\ \text{if } a_2 = 0 \\ neg \\ \text{otherwise} \end{cases}$	$\mathcal{A}op_{strict}[\![div]\!]\ a_1\ a_2$

$*$ a_1 / a_2	neg	Val
neg	$\mathcal{A}op_{neg}[\![*]\!]\ a_1\ a_2$	$\begin{cases} 0 \\ \text{if } a_1 = 0 \\ neg \\ \text{otherwise} \end{cases}$
Val	$\begin{cases} 0 \\ \text{if } a_2 = 0 \\ neg \\ \text{otherwise} \end{cases}$	$\mathcal{A}op_{strict}[\![*]\!]\ a_1\ a_2$

mod a_1 / a_2	neg	Val
neg	$error$	$error$
Val	$error$	$\mathcal{A}op_{strict}[\![mod]\!]\ a_1\ a_2$

Table 5.6: Arithmetic operators semantics

Val, the comparison results from the concrete comparison between integer values as it is detailed in Table 5.7.

$$\mathcal{C}op_{strict} : Cop \rightarrow Val \rightarrow Val \rightarrow \{true, false\}$$

Boolean operators $\mathcal{B}op$:
$Bop{+}Bopltr \rightarrow \{true, false, error\} \rightarrow \{true, false, error\} \rightarrow \{true, false, error\}$
Table 5.10 contains the semantics of each operator from the Bop and $Bopltr$. For the operators $bop \in Bop$, $error$ is propagated when a term is $error$, the propagation is similar to the one of an arithmetic evaluation or a comparison evaluation.

However, for operators $bopltr \in Bopltr$, the order of the evaluation is important: let us have a look at the operator &&; if the first expression is evaluated to false, the result is false, and the second expression does not have to be evaluated. In other terms, it does not matter if the evaluation of the second expression generates an error or not.

5.2.2.8 Statements

Let us define the semantics of the statements; the execution of a statement updates the environment. An assignment command accesses and changes the environment. A conditional statement transforms the environment according to the environment state: a Boolean expression is evaluated in this environment to know in which way this will be transformed. A subproblem call transforms the environment according to the interpretation set *itp*. A sequence is interpreted as a composition of environment-transforming functions and errors are propagated. The skip command has no effect on the environment.

$$\mathcal{C}op_{strict} : Cop \rightarrow Val \rightarrow Val \rightarrow \{true, false\}$$

$\mathcal{C}op_{strict}\llbracket=\rrbracket\ i_1\ i_2 = i_1 = i_2$
$\mathcal{C}op_{strict}\llbracket!=\rrbracket\ i_1\ i_2 = i_1 \neq i_2$
$\mathcal{C}op_{strict}\llbracket<\rrbracket\ i_1\ i_2 = i_1 < i_2$
$\mathcal{C}op_{strict}\llbracket>\rrbracket\ i_1\ i_2 = i_1 > i_2$
$\mathcal{C}op_{strict}\llbracket<=\rrbracket\ i_1\ i_2 = i_1 \leq i_2$
$\mathcal{C}op_{strict}\llbracket>=\rrbracket\ i_1\ i_2 = i_1 \geq i_2$

Table 5.7: Semantics of relational operators between values Val

$$\mathcal{C}op : Cop \rightarrow Val{+}\{neg, error\} \rightarrow Val{+}\{neg, error\} \rightarrow \{true, false, error\}$$

cop a_1 / a_2	$error$	$Val + \{neg\}$
$error$	$error$	$error$
$Val + \{neg\}$	$error$	$\mathcal{C}op_A\llbracket cop\rrbracket\ a_1\ a_2$

Table 5.8: Error propagation

$$\mathcal{C}op_A : Cop \rightarrow Val + \{neg\} \rightarrow Val + \{neg\} \rightarrow \{true, false, error\}$$

$<$ a_1 / a_2	neg	Val
neg	$error$	$false$
Val	$true$	$\mathcal{C}op_{strict}\llbracket<\rrbracket\ a_1\ a_2$

$<=$ a_1 / a_2	neg	Val
neg	$error$	$false$
Val	$true$	$\mathcal{C}op_{strict}\llbracket<=\rrbracket\ a_1\ a_2$

$>$ a_1 / a_2	neg	Val
neg	$error$	$true$
Val	$false$	$\mathcal{C}op_{strict}\llbracket>\rrbracket\ i_1\ i_2$

$>=$ a_1 / a_2	neg	Val
neg	$error$	$true$
Val	$false$	$\mathcal{C}op_{strict}\llbracket>=\rrbracket\ a_1\ a_2$

$=$ a_1 / a_2	neg	Val
neg	$error$	$false$
Val	$false$	$\mathcal{C}op_{strict}\llbracket=\rrbracket\ a_1\ a_2$

$!=$ a_1 / a_2	neg	Val
neg	$error$	$true$
Val	$true$	$\mathcal{C}op_{strict}\llbracket!=\rrbracket\ a_1\ a_2$

Table 5.9: Semantics of Relational operators

$$\mathcal{C} : Com \to Itp \to Env_{td} \to Env_{td} + \{error, \bot\}$$

$$\begin{array}{lcl}
\mathcal{C}[\![skip]\!]\ itp\ e & = & e \\
\mathcal{C}[\![x := aexpr]\!]\ itp\ e & = & e[x/\mathcal{E}val[\![aexpr]\!]\ e] \\
 & & \text{if } \mathcal{E}val[\![aexpr]\!]\ e\ \in td_{id}(x) \\
 & & error \text{ otherwise} \\
\mathcal{C}[\![x := bexpr]\!]\ itp\ e & = & e[x/\mathcal{B}[\![bexpr]\!]\ e] \\
 & & \text{if } \mathcal{B}[\![bexpr]\!]\ e \neq error \\
 & & error \text{ otherwise} \\
\mathcal{C}[\![tab[aexpr_1] := aexpr_2]\!]\ itp\ e & = & e[tab(\mathcal{E}val[\![aexpr_1]\!]\ e)/\mathcal{E}val[\![aexpr_2]\!]\ e] \\
 & & \text{if } \mathcal{E}val[\![aexpr_1]\!]\ e \in dom(e_{tab}(tab)) \\
 & & \text{and } \mathcal{E}val[\![aexpr_2]\!]\ e \in d \\
 & & \text{where } (l,d) = td_{tab}(tab) \\
 & & error \text{ otherwise} \\
\mathcal{C}[\![\textbf{if}\ bexpr\ \textbf{then}\ com_1\ \textbf{else}\ com_2]\!]\ itp\ e & = & \mathcal{C}[\![com_1]\!]\ itp\ e \text{ if } \mathcal{B}[\![bexpr]\!]\ e = true \\
 & & \mathcal{C}[\![com_2]\!]\ itp\ e \text{ if } \mathcal{B}[\![bexpr]\!]\ e = false \\
 & & error \text{ otherwise} \\
\mathcal{C}[\![p]\!]\ itp\ e & = & itp(p)\ e \\
\mathcal{C}[\![com_1; com_2]\!]\ itp\ e & = & \mathcal{C}[\![com_2]\!]\ itp\ e' \\
 & & \text{with } e' = \mathcal{C}[\![com_1]\!]\ itp\ e \in Env_{td} \\
 & & error \text{ otherwise}
\end{array}$$

5.2.2.9 Subproblems and program

Subproblem Given a static environment td, the semantics of a subproblem is the transformation of the set of environments satisfying td. These transformations may generate an error or a loop. We define the function $\mathcal{SP}$ that, according to a set of interpretations $itp \in Itp$, defines the semantics of a subproblem.

$$\mathcal{SP} : Body \to Itp \to (Env_{td} \to Env_{td} + \{error, \bot\})$$

$\mathcal{SP}[\![com]\!]\ itp = f$

$$\begin{array}{lll}
\text{where } f : & Env_{td} & \to Env_{td} + \{error, \bot\} \\
 & e & \to \mathcal{C}[\![com]\!]\ itp\ e
\end{array}$$

The semantics of a simple subproblem, according to the set of interpretations itp is a function mapping a set of environments e_{in} into a set of environments e_{out} (if no termination problem occurs). Each of these environments satisfies the domains specified in td. This function is defined from the semantics of the statements.

$\mathcal{SP}[\![com_1\ com_2\ com_3\ bexpr]\!]\ itp = f$

$$\begin{array}{llll}
\text{where } f : & Env_{td} & \to Env_{td} + \{error, \bot\} & \\
 & e & \to f_3(e_2) & \text{if } e_2 \in Env_{td} \\
 & & \ \ \ e_2 & \text{otherwise}
\end{array}$$

$$\begin{array}{l} \text{where } e_2 = f_2(e_1) \text{ if } e_1 \in Env_{td} \\ \qquad\quad e_2 = e_1 \qquad \text{otherwise} \\ \qquad\quad \text{where } e_1 = f_1(e) \end{array}$$

where
$f_1 = \mathcal{C}[\![com_1]\!]\ itp$
$f_2 = loop(com_2, bexpr)\ itp$
$f_3 = \mathcal{C}[\![com_3]\!]\ itp$

$loop : Com \times Bexpr \to Itp \to Env_{td} \to Env_{td} + \{error, \bot\}$

$$\begin{array}{lcll} loop(com, bexpr)\ itp\ e & = & e_n & \text{if } n < \infty \\ & & \bot & \text{if } n = \infty \end{array}$$

where e_n is the first e_i in L such that $e_n = e_n + 1$.
L is the list $e_1, e_2, ...e_n, e_{n+1}, \ldots$ where all $e_i \in Env_{td} + \{error, \bot\}$;
this list is defined as follows:

$$\begin{cases} e_1 & = & e & \\ e_{i+1} & = & e_i & \text{if } e_i \notin Env_{td} \text{ or else } \mathcal{B}[\![bexpr]\!]\ e_i = true \\ & & \mathcal{C}[\![com]\!]\ itp\ e_i & \text{otherwise} \end{cases}$$

For a subproblem with a loop form (three statements blocks and a halting condition), we have a composition of environment-transforming functions f_1, f_2, f_3. *error* and $\bot$ are propagated. f_2 is the result of the function *loop* which defines an iteration of the environment transformation: expression *bexpr* is evaluated and command com_2 is executed repeatedly, alternating, until the value of *bexpr* becomes *true*. Unfortunately, it is possible that the value of *bexpr* never becomes *true*, then, the execution does not terminate. $\bot$ is an undefined environment due to a command that fails to terminate.

Program A program is defined as a set of subproblems semantically defined according to a static environment *td*. A program is represented by a function $itp \in Itp$ defined in section 5.2.2.4, so that $dom(itp)$ is the set of the identifiers of the subproblems composing the program. The function *itp* is defined as follows:

$$\mathcal{P}rog = Prog \to Itp$$

$$\mathcal{P}rog[\![p : spr]\!] = \phi[p \mapsto \mathcal{SP}[\![body]\!]\ \phi] \quad \text{where } spr = decl\ body$$

$$\begin{array}{l} \mathcal{P}rog[\![p : spr\ (p_i : spr_i)^+]\!] = itp'[p \mapsto \mathcal{SP}[\![body]\!]\ itp'] \\ \qquad\qquad \text{where } spr = decl\ body \\ \qquad\qquad \text{where } itp' = \mathcal{P}rog[\![(p_i : spr_i)^+]\!] \end{array}$$

Notice that to update the interpretation set, we begin by defining the subproblems lying on the right in the list $p_1 : spr_1\ (p : spr)^*$. Indeed, by convention, the called subproblems must lie on the right of the subproblem calling and to define the semantics of a problem we need the semantics of the subproblems called.

5.3 The Assertion Language

5.3.1 Abstract Syntax

First, in Table 5.11, we extend syntactical domains and generic variables from Table 5.1. Table 5.12 specifies the *syntax* of the assertion language. All the operators such as arithmetic, comparison and Boolean operators of the programming language are admitted in the assertion language. Some expressions use extensions as x_0 or tab_0 which is a way of referring resp. to the value of the variable x at the precondition state and to the array *tab* with its initial values. Other logical operators also are available, such as implication `=>` and equivalence `<=>`. Importantly, this language supports several kinds of quantifiers in assertions: the *universal quantifier*, the *existential quantifier*, the *generalised quantifiers* `sum`, `min` and `max` and the *numeric quantifier* `#`. Besides, to make our assertion language as convenient as possible, we have some predefined predicates such as the unchanged predicate, the permutation predicate. The comparison operators between arrays to express the lexicographic order also exist.

5.3.2 Semantics

As for the programming language, the expression evaluation function consults the environment but does not modify it. We use the same semantic domains as those we have defined in Section 5 (see Figure 5.1 and 5.2).

There are three main observations:

- An assertion or a generalised numerical expression is evaluated according to a particular static environment *td* (which is usually initialised by the program we want to check)

  For instance, we need to know the size of arrays to check out-of-bound errors.

- The evaluation function may consult two environments:

  - the environment at the program point where we evaluate the expression, say $e \in Env_{td}$,
  - the environment at the precondition point, named $e_0 \in Env_{td}$:

    we may express something according to the initial value of a variable.

- Bodies of quantified expressions are evaluated according to a *td* which is updated with the bounded variables of the quantified expression.

$\&$ b_1 / b_2	$true$	$false$	$error$
$true$	$true$	$false$	$error$
$false$	$false$	$false$	$error$
$error$	$error$	$error$	$error$

$\&\&$ b_1 / b_2	$true$	$false$	$error$
$true$	$true$	$false$	$error$
$false$	$false$	$false$	$error$
$error$	$error$	$false$	$error$

$\mid$ b_1 / b_2	$true$	$false$	$error$
$true$	$true$	$true$	$error$
$false$	$true$	$false$	$error$
$error$	$error$	$error$	$error$

$\mid\mid$ b_1 / b_2	$true$	$false$	$error$
$true$	$true$	$true$	$error$
$false$	$true$	$false$	$error$
$error$	$true$	$error$	$error$

$=$ b_1 / b_2	$true$	$false$	$error$
$true$	$true$	$false$	$error$
$false$	$false$	$true$	$error$
$error$	$error$	$error$	$error$

$!=$ b_1 / b_2	$true$	$false$	$error$
$true$	$false$	$true$	$error$
$false$	$true$	$false$	$error$
$error$	$error$	$error$	$error$

Table 5.10: Boolean operators Semantics

$assert \in Assert$	assertions
$eaexpr \in Eaexpr$	*extended* numeral expressions
$gaexpr \in GAexpr$	*generalised* numeral expressions
$dom \in Dom$	bounded domains
$iop \in Implop$	impliement operators
$taop \in TAop$	numeral quantifier operators
$tbop \in TBop$	Boolean quantifier operators
$tcop \in TCop$	relation operators between arrays

Table 5.11: Syntax Domains, Generic Variables for assertions

Assertions

$$
\begin{array}{lll}
\mathit{assert} & ::= & \mathit{bexpr} \\
 & | & \mathit{gaexpr}\ \mathit{cop}\ \mathit{gaexpr} \\
 & | & \mathit{gtab}\ \mathit{tcop}\ \mathit{gtab} \\
 & | & \mathit{assert}\ \mathit{bop}\ \mathit{assert} \\
 & | & \mathit{assert}\ \mathit{iop}\ \mathit{assert} \\
 & | & \mathit{bnot}\ \mathit{assert} \\
 & | & \textbf{forall}\ x : \mathit{dom} : \mathit{assert} \\
 & | & \textbf{exist}\ x : \mathit{dom} : \mathit{assert} \\
 & | & \textbf{forall}\ \mathit{tab}[1..\mathit{eaexpr}] : \mathit{dom} : \mathit{assert} \\
 & | & \textbf{exist}\ \mathit{tab}[1..\mathit{eaexpr}] : \mathit{dom} : \mathit{assert} \\
 & | & \textbf{permut}\ (\mathit{dom}, \mathit{gtab}, \mathit{dom}, \mathit{gtab}) \\
 & | & \textbf{unchanged}\ x \\
 & | & \textbf{unchanged}\ (\mathit{dom}, \mathit{gtab}) \\
\mathit{dom} & ::= & \mathit{eaexpr}..\mathit{eaexpr}
\end{array}
$$

Extended Numerical Expressions

$$
\begin{array}{lll}
\mathit{eaexpr} & ::= & \mathit{aexpr} \\
 & | & x_0 \\
 & | & \mathit{tab}[\mathit{eaexpr}] \\
 & | & \mathit{tab}_0[\mathit{eaexpr}] \\
 & | & \mathit{eaexpr}\ \mathit{aop}\ \mathit{eaexpr}
\end{array}
$$

Generalised Numerical Expressions

$$
\begin{array}{lll}
\mathit{gaexpr} & ::= & \mathit{eaexpr} \\
 & | & \mathit{gaexpr}\ \mathit{aop}\ \mathit{gaexpr} \\
 & | & \mathit{gaexpr}\ \mathit{exp}\ \mathit{gaexpr} \\
 & | & \mathit{taop}\ x : \mathit{dom} : \mathit{gaexpr} \\
 & | & \mathit{tbop}\ x : \mathit{dom} : \mathit{assert} \\
\mathit{gtab} & ::= & \mathit{tab}_0 \mid \mathit{tab}
\end{array}
$$

Operators

$$
\begin{array}{lll}
\mathit{iop} & ::= & \texttt{=>} \mid \texttt{<=>} \\
\mathit{taop} & ::= & \textbf{sum} \mid \textbf{max} \mid \textbf{min} \\
\mathit{tbop} & ::= & \# \\
\mathit{tcop} & ::= & = \mid\ != \mid\ << \mid\ =<< \mid\ >> \mid\ >>=
\end{array}
$$

Table 5.12: Assertion syntax

5.3.2.1 Evaluation of generalised numerical expressions

The evaluation of a generalised numerical expression should result into a numerical value, but it can also generate an error because, for example, of an array out-of-bound error, of a division by zero or simply because the variable to be evaluated is not initialised. If we have enough information to testify that the result is negative, the evaluation result is an "abstract" value, *neg*. If we do not have enough information to testify that the result is negative or has a precise numerical value, the result of the evaluation is *error*.

$$\mathcal{G}eval_{td} : GAexpr \rightarrow Env_{td} \rightarrow Env_{td} \rightarrow Val + \{neg, error\}$$

$$\begin{array}{lcl}
\mathcal{G}eval_{td}[\![aexpr]\!]\ e_0\ e & = & \mathcal{E}val[\![aexpr]\!]\ e \\
\mathcal{G}eval_{td}[\![x_0]\!]\ e_0\ e & = & e_{0_{id}}(x) \text{ if } e_{0_{id}}(x) \neq noval \\
 & & error \text{ otherwise} \\
\mathcal{G}eval_{td}[\![tab[eaexpr]]\!]\ e_0\ e & = & e_{tab}(tab)(\mathcal{G}eval_{td}[\![eaexpr]\!]\ e_0\ e) \\
 & & \text{if } \mathcal{G}eval_{td}[\![eaexpr]\!]\ e_0\ e \in dom(e_{tab}(tab)) \\
 & & error \text{ otherwise} \\
\mathcal{G}eval_{td}[\![tab_0[eaexpr]]\!]\ e_0\ e & = & e_{0_{tab}}(tab)(\mathcal{G}eval_{td}[\![eaexpr]\!]\ e_0\ e) \\
 & & \text{if } \mathcal{G}eval_{td}[\![eaexpr]\!]\ e_0\ e \in dom(e_{tab}(tab)) \\
 & & error \text{ otherwise} \\
\mathcal{G}eval_{td}[\![gaexpr\ aop\ gaexpr]\!]\ e_0\ e & = & \mathcal{A}op[\![aop]\!]\ (i_1, i_2) \\
 & & \text{with } i_1 = \mathcal{G}eval_{td}[\![gaexpr_1]\!]\ e_0\ e \\
 & & \text{and } i_2 = \mathcal{G}eval_{td}[\![gaexpr_2]\!]\ e_0\ e \\
\mathcal{G}eval_{td}[\![gaexpr_1\ \mathbf{exp}\ gaexpr_2]\!]\ e_0\ e & = & i_1^{i_2} \text{ if } i_1^{i_2} \in Val, \\
 & & \text{with } i_1 = \mathcal{G}eval_{td}[\![gaexpr_1]\!]\ e_0\ e \in Val \\
 & & \text{and } i_2 = \mathcal{G}eval_{td}[\![gaexpr_2]\!]\ e_0\ e \in Val \\
 & & error \text{ otherwise}
\end{array}$$

The generalised numeral expressions may refer to the value of a variable at the precondition state: we consult this value through the environment of the precondition, e_0. $\mathcal{A}op$ is the semantic function of the operators $aop \in Aop$ described in Table 5.3.

In a quantified formula, there is a declaration of a variable local to the quantifier (`x`). This is followed by a range (*dom*) on a finite domain for which the bounds are extended expressions (*eaexpr*) and a body (*gaexpr*). If the range is badly defined, i.e., one of the bounds is badly defined, the evaluation of the quantified expression generates an error.

Assume $d = \mathcal{D}om_{td}[\![dom]\!]\ e_0\ e \neq error$
Let $a_i = \mathcal{G}eval_{td[x/d]}[\![gaexpr]\!]\ e_0\ e[x/i]$

$$\begin{array}{lcl}
\mathcal{G}eval_{td}([\![taop\ x : dom : gaexpr]\!]\ e_0\ e & = & \mathcal{T}a[\![taop]\!]\ f \\
 & & \text{where } f : N \nrightarrow Val + \{neg, error\} \\
 & & \text{with } dom(f) = d \\
 & & \text{and forall } i \in dom(f) : f(i) = a_i
\end{array}$$

`sum`, `min` and `max` return respectively the sum, the minimum and the maximum of the value of the body when the quantified variables satisfy the given range expression. To compute it:

- We evaluate the range on finite domain with the function $\mathcal{D}om$ which is described below.
- Then, we evaluate *gaexpr* for each value i satisfying the range expression (to achieve it, we update the environment by adding the local declaration and giving it the value i).
- If no error is found in the evaluation of each body *geaxpr*, the function $\mathcal{T}a$ computes the sum, the minimum or the maximum of the set of *geaexpr* evaluation result;

  otherwise, the error is propagated.

Similar computing is done for the numeric quantifier # which returns the number of values for the quantified variable for which the range and the body predicate are true:

$$\begin{array}{lcl}
\text{Assume } d = \mathcal{D}om_{td}[\![dom]\!]\ e_0\ e \neq error & & \\
\text{Let } a_i = \mathcal{A}_{td[x/d]}[\![assert]\!]\ e_0\ e[x/i] & & \\
\mathcal{G}eval_{td}[\![tbop\ x : dom : assert]\!]\ e_0\ e & = & \mathcal{T}b[\![\#]\!]\ f \\
 & & \text{where } f : N \nrightarrow Bool + \{error\} \\
 & & \text{with } dom(f) = d \\
 & & \text{and forall } i \in dom(f) : f(i) = a_i
\end{array}$$

The function $\mathcal{D}om_{td}$ gives the semantics of $dom \in Dom$, it returns a interval on which we can evaluate the quantified expression. It is in this context that we observe a reason why we define *neg* in our abstract domain. Let $[i_1..i_2]$ be an interval, if $i_1 \in Val$ and i_2 is negative ($i_2 = neg$), no matter the concrete value of i_2: the interval $[i_1..i_2]$ is empty. However, an interval where i_1 is negative is not manageable in Oz (and so, by our tool).

$$\mathcal{D}om : Dom \rightarrow Env_{td} \rightarrow Env_{td} \rightarrow D + \{error\}$$

$$\begin{array}{lcll}
\multicolumn{4}{l}{\text{Assume } i_1 = \mathcal{G}eval_{td}[\![eaexpr_1]\!]\ e_0\ e \text{ and } i_2 = \mathcal{G}eval_{td}[\![eaexpr_2]\!]\ e_0\ e} \\
\mathcal{D}om_{td}[\![eaexpr_1 \leq x \leq eaexpr_2]\!]\ e_0\ e & = & [i_1..i_2] & \text{if } i_1 \in Val \text{ and } i_2 \in Val \\
 & & [] & \text{if } i_1 \in Val \text{ and } i_2 = neg \\
 & & error & \text{otherwise}
\end{array}$$

$\mathcal{T}a$ is defined in Table 5.13 and gives the semantics of the generalised arithmetic operators: **sum** computes a sum of the elements of a set, **max** and **min** returns resp. the maximal and the minimal element of a set defined through a partial function $f : N \nrightarrow Val + \{neg, error\}$ where $dom(f)$ is range on finite domain. The generalised arithmetic operators do not manage negative value[2]; it means that the evaluation for each value $i \in dom(f)$ must be in Val; otherwise, the evaluation generates an error. Errors are also propagated if at least one of the evaluations is $error$. Notice that for **max** and **min**, $dom(f)$ cannot be empty.

The operators $tbop \in Tbop$ is defined in Table 5.14; the partial function that we use is $f : N \nrightarrow Bool + \{error\}$. `#` computes the number of values of $dom(f)$ such that $f(i) = true$. Errors are also propagated if one of the evaluation results in an error.

5.3.2.2 Evaluation of assertions

To define the semantics of the assertion, we divide the set of assertions into three sets. First we consider the binary and unary operations, containing Boolean operators and relational operators. Then, we consider quantified assertions, and finally, we define some useful predicate. Anyway, the evaluation of an assertion returns $true$ or $false$ or it may generate an error:

$$\mathcal{A}_{td} : Assert \rightarrow Env_{td} \rightarrow Env_{td} \rightarrow \{true, false, error\}$$

Binary and unary operations The semantics of the relational operators and Boolean operators are given in Table 5.9 and Table 5.10. Two other Boolean operators are available: the implication and the equivalence. Table 5.15 defines the semantics of these operators. We have also defined the lexicographic order between arrays. The definition is given in Table 5.16. In order to make operation on arrays, we define the following function:

$$\mathcal{G} : Tab + Tab_0 \rightarrow \ Env_{td} \ \rightarrow \ Env_{td} \rightarrow (N \nrightarrow Val)$$

$\mathcal{G}_{td}[\![tab_0]\!]\ e_0\ e = e_0(tab)$
$\mathcal{G}_{td}[\![tab]\!]\ e_0\ e = e(tab)$
The function $\mathcal{G}$ gives the state of an array tab according to the current environment or to the precondition environment.

[2]We could have chosen to accept negative values, because in some cases abstract interpretation could give a precise result.

$$\mathcal{T}a : Taop \rightarrow (N \nrightarrow Val + \{neg, error\}) \rightarrow Val + \{error\}$$

$$\begin{array}{lcll}
\mathcal{T}a[\![\mathbf{sum}]\!]\ f & = & \sum_{i \in dom(f)} f(i) & \text{if } codom(f) \subseteq Val \\
 & & & \text{and } \sum_{i \in d} f(i) \in Val \\
 & & error & \text{otherwise} \\
\mathcal{T}a[\![\mathbf{max}]\!]\ f & = & f(i) & \text{with } i \in dom(f) \\
 & & & \text{and forall } j \in dom(f) : f(i) \geq f(j) \\
 & & & \text{if } codom(f) \subseteq Val \\
 & & & \text{and } dom(f) \neq [] \\
 & & error & \text{otherwise} \\
\mathcal{T}a[\![\mathbf{min}]\!]\ f & = & f(i) & \text{with } i \in dom(f) \\
 & & & \text{and forall } j \in dom(f) : f(i) \leq f(j) \\
 & & & \text{if } codom(f) \subseteq Val \\
 & & & \text{and } dom(f) \neq [] \\
 & & error & \text{otherwise}
\end{array}$$

Table 5.13: Semantics of the generalised arithmetic operators

$$\mathcal{T}b : Tbop \rightarrow (N \nrightarrow Bool + \{error\}) \rightarrow Val + \{error\}$$

$$\begin{array}{lcll}
\mathcal{T}b[\![\#]\!]\ f & = & \sum_{i \in dom(f)} g(f(i)) & \text{with } g : Bool \rightarrow \{0,1\} \\
 & & & \text{such that } g(true) = 1 \text{ and } g(false) = 0 \\
 & & & \text{if } codom(f) \subseteq Bool \\
 & & error & \text{otherwise}
\end{array}$$

Table 5.14: Semantics of the generalised arithmetic operator #

$$\mathcal{A}_{td} : Assert \rightarrow Env_{td} \rightarrow Env_{td} \rightarrow \{true, false, error\}$$

$$\begin{array}{lcl}
\mathcal{A}_{td}[\![bexpr]\!]\ e_0\ e & = & \mathcal{B}[\![bexpr]\!]\ e \\
\mathcal{A}_{td}[\![gaexpr_1\ cop\ gaexpr_2]\!]\ e_0\ e & = & \mathcal{C}op[\![bop]\!]\ (i_1, i_2) \\
 & & \text{with } i_1 = \mathcal{G}eval_{td}[\![gaexpr_1]\!]\ e_0\ e \\
 & & \text{and } i_2 = \mathcal{G}eval_{td}[\![gaexpr_2]\!]\ e_0\ e \\
\mathcal{A}_{td}[\![assert_1\ bop\ assert_2]\!]\ e_0\ e & = & \mathcal{B}op[\![bop]\!]\ (b_1, b_2) \\
 & & \text{with } b_1 = \mathcal{A}_{td}[\![assert_1]\!]\ e_0\ e \\
 & & \text{and } b_2 = \mathcal{A}_{td}[\![Assert_2]\!]\ e_0\ e \\
\mathcal{A}_{td}[\![assert_1\ bopltr\ assert_2]\!]\ e_0\ e & = & \mathcal{B}op[\![bopltr]\!]\ (b_1, b_2) \\
 & & \text{with } b_1 = \mathcal{A}_{td}[\![assert_1]\!]\ e_0\ e \\
 & & \text{and } b_2 = \mathcal{A}_{td}[\![Assert_2]\!]\ e_0\ e \\
\mathcal{A}_{td}[\![bnot\ assert]\!]\ e_0\ e & = & true \text{ if} \mathcal{A}_{td}[\![assert]\!]\ e_0\ e = false \\
 & & false \text{ if} \mathcal{A}_{td}[\![assert]\!]\ e_0\ e = true \\
 & & error \text{ otherwise} \\
\mathcal{A}_{td}[\![assert_1\ iop\ assert_2]\!]\ e_0\ e & = & \mathcal{I}op[\![iop]\!]\ (b_1, b_2) \\
 & & \text{with } b_1 = \mathcal{A}_{td}[\![assert_1]\!]\ e_0\ e \\
 & & \text{and } b_2 = \mathcal{A}_{td}[\![Assert_2]\!]\ e_0\ e \\
\text{Let } f_1 = \mathcal{G}_{td}[\![gtab_1]\!]\ e_0\ e & & \\
\text{Let } f_2 = \mathcal{G}_{td}[\![gtab_2]\!]\ e_0\ e & & \\
\text{Assume } [1..l_1] = dom(f_1) & & \\
\text{and } [1..l_2] = dom(f_2) & & \\
\mathcal{A}_{td}[\![gtab_1\ tcop\ gtab_1]\!]\ e_0\ e & = & false \text{ if } l1 \neq l_2 \\
 & & \mathcal{T}cop[\![tcop]\!]\ f_1\ f_2 \text{ otherwise}
\end{array}$$

Quantified expressions In a quantified assertion, there is a declaration of a local variable (x), there is a range on finite domain (*dom*) and the body (*assert*). If the domain *dom* is badly defined, the evaluation generates an error; otherwise, with a universal quantifier, the body must be true for all the values that satisfy the given range expression; with an existential quantifier, the body must be true for at least one value that satisfies the given range expression. In both cases, if the evaluation of the body generates an error for one value, this error is propagated and the evaluation of the quantified assertion is *error*.

$$\begin{array}{lcll}
\multicolumn{4}{l}{\text{Assume } d = \mathcal{D}om_{td}[\![dom]\!]\ e_0\ e \neq error} \\
\multicolumn{4}{l}{\text{Let } b_k = \mathcal{A}_{td[x/d]}[\![assert]\!]\ e_0\ e_{id}[x/k]\ (k \in d)} \\
\mathcal{A}_{td}[\![\textbf{forall}\ x : dom : assert]\!]\ e_0\ e & = & error & \text{if } b_k = error \text{ for some } k \in d \\
 & & true & \text{if } b_k = true \text{ for all } k \in d \\
 & & false & \text{otherwise} \\
\mathcal{A}_{td}[\![\textbf{exist}\ x : dom : assert]\!]\ e_0\ e & = & error & \text{if } b_k = error \text{ for some } k \in d \\
 & & false & \text{if } b_k = false \text{ for all } k \in d \\
 & & true & \text{otherwise}
\end{array}$$

It is also possible to have quantified assertions whose local variable is an array $a[1..ea]$ where $ea \in Eaexpr$:

$=>$ b_1 b_2	*true*	*false*	*error*
true	*true*	*true*	*error*
false	*false*	*true*	*error*
error	*error*	*error*	*error*

$<=>$ i_1 i_2	*true*	*false*	*error*
true	*true*	*false*	*error*
false	*false*	*true*	*error*
error	*error*	*false*	*error*

Table 5.15: Implication and equivalence operators Semantics

$$\mathcal{T}cop : Tcop \rightarrow (N \nrightarrow Val) \rightarrow (N \nrightarrow Val) \rightarrow \{true, false\}$$

Assume $[1..l] = dom(f_1) = dom(f_2)$

$$\begin{array}{llll}
\mathcal{T}cop(\succeq)\ f_1\ f_2 & = & true & \text{if } f_1 = f_2 \\
 & & & \text{or for some } i \in [1..l-1], \\
 & & & f_1(j) = f_2(j)\text{for all } j \in [1..i-1] \\
 & & & \text{and } f_1(i) > f_2(i) \\
 & & false & \text{otherwise} \\
\mathcal{T}cop(\succ)\ f_1\ f_2 & = & true & \text{if for some } i \in [1..l-1], f_1(i) > f_2(i), \\
 & & & \text{and } f_1(j) = f_2(j) \text{ for all } j \in [1..i-1] \\
 & & false & \text{otherwise} \\
\mathcal{T}cop(\preceq)\ f_1\ f_2 & = & true & \text{if } f_1 = f_2 \\
 & & & \text{or for some } i \in [1..l-1], f_1(i) < f_2(i) \\
 & & & \text{and } f_1(j) = f_2(j)\text{for all } j \in [1..i-1] \\
 & & false & \text{otherwise} \\
\mathcal{T}cop(\prec)\ f_1\ f_2 & = & true & \text{if for some } i \in [1..l-1], f_1(i) < f_2(i) \\
 & & & \text{and } f_1(j) = f_2(j) \text{ for all } j \in [1..i-1] \\
 & & false & \text{otherwise}
\end{array}$$

Table 5.16: Semantics of the lexicographic relation operators

Assume $d = \mathcal{D}om_{td}[\![dom]\!]\ e_0\ e \neq error$
Assume $l = \mathcal{G}eval_{td}[\![ea]\!]\ e_0\ e \neq error$
Let $T = \{f : N \nrightarrow Val | dom(f) = [1..l] \text{ and } codom(f) \subseteq d\}$
Let $b_f = \mathcal{A}_{td[tab/(l,d)]}[\![assert]\!]\ e_0\ e_{id}[tab/f]\ (f \in T)$

$$\begin{array}{lcll}
\mathcal{A}_{td}[\![\textbf{forall}\ tab[1..ea] : dom : assert]\!]\ e_0\ e & = & error & \text{if } b_f = error \text{ for some } f \in T \\
 & & true & \text{if } b_f = true \text{ for all } f \in T \\
 & & false & \text{otherwise} \\
\mathcal{A}_{td}[\![\textbf{exist}\ tab[1..ea] : dom : assert]\!]\ e_0\ e & = & error & \text{if } b_f = error \text{ for some } f \in T \\
 & & true & \text{if } b_f = true \text{ for all } f \in T \\
 & & false & \text{otherwise}
\end{array}$$

Predefined predicates The predicate **unchanged** compares the value of a variable with its initial value; it can also compare the values of an array with its initial values.

$$\begin{array}{lcll}
\mathcal{A}_{td}[\![\textbf{unchanged}\ x]\!]\ e_0\ e & = & e_0(x) = e(x) & \text{if } e_0(x) \neq noval \\
 & & error & \text{otherwise}
\end{array}$$

Assume $d = \mathcal{D}om_{td}[\![dom]\!]\ e_0\ e \neq error$

$$\begin{array}{lcll}
\mathcal{A}_{td}[\![\textbf{unchanged}(dom, tab)]\!]\ e_0\ e & = & true & \text{if } e_0(tab)(k) = e(tab)(k) \text{ for all } k \in d \\
 & & error & \text{if for some } k \in d : k \notin dom(e_0(tab)) \\
 & & false & \text{otherwise}
\end{array}$$

The predicate **permut** expresses that the set of values of two subarrays corresponds to the same set of values with identical duplications if any: there is a bijection f between the indexes of the two subarrays, let tab_1 and tab_2, such that $tab_1[k] = tab_2[f(k)]$ for index k of the subarray tab_1:

Assume $d_1 = \mathcal{D}om_{td}[\![dom_1]\!]\ e_0\ e \neq error$
and $d_2 = \mathcal{D}om_{td}[\![dom_2]\!]\ e_0\ e \neq error$
Assume $f_1 = \mathcal{G}_{td}[\![gtab_1]\!]\ e\ e_0$
and $f_2 = \mathcal{G}_{td}[\![gtab_2]\!]\ e\ e_0$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{A}_{td}[\![\textbf{permut}\ (dom_1, gtab_1, dom_2, gtab_2)]\!]\ e_0\ e} \\
 & = true & \text{if } \#d_1 = \#d2 \\
 & & \text{and for some } f \in Bij(d_1, d_2) : f_2 \circ f = f_1 \\
 & = false & \text{otherwise}
\end{array}$$

where $Bij(d_1, d_2)$ is the set of bijections between the two sets d_1 and d_2; $\#d_i$ expresses the size of d_i (cardinality).

Chapter 6

Implementation of MPVS using the Mozart Programming System

6.1 Introduction

In Chapter 4, we have observed that MPVS is able to prove the correctness of our algorithms. This chapter will hopefully reveal some of the mysteries.

Constraint programming over finite domains seems to us especially convenient to check the kind of verification conditions that are needed to express the correctness of imperative programs. We have chosen the multiparadigm language Oz because to conveniently generate the constraint problems equivalent to a given verification condition, it is desirable to have at hand a powerful language that allows us to interleave constraint generation and constraint solving. Oz includes all programming mechanisms that are needed to reach our goals. In addition, we have an excellent support because we are working in UCL, in the same department as the Belgian Mozart/Oz pool which is partially responsible for the implementation of the Oz language. We also have been easily convinced to use it.

Chapter 6 focuses on the way our programs are automatically translated into constraints, and how the constraints are solved. Most explanations have already been published in [18, 20]. We also show how the violated parts of an assertion can be precisely provided to the user. We show that the architecture of MPVS is very simple.

In Section 6.2, we provide an overview of the Oz programming language and the Mozart system that could be useful to understand this chapter.

Then, we explain in Section 6.3 the use of verification conditions to check the correctness of our programs. We introduce an example of verification condition that we follow through the next sections: we start from a simple translation into constraints and then, gradually, we add some technical aspects too difficult to assimilate in the first place. So, in Section 6.4 and Section 6.5, we manually translate a subset of the verification condition example and we solve this constraint problem. We give a first idea of the heuristics of the distribution strategy. In Section 6.6, we explain the automatic translation of our assertions into a set of constraints. In Section 6.7, we motivate the interleaving between constraint generation and constraint solving. We argue that our method is complete in Section 6.8. More complex techniques in our implementation are introduced in Section 6.9: we consider the detection of runtime errors and of incorrect assertions, and we deal with negative expressions; with these final improvements, our method is sound. Section 6.10 displays the simple architecture of the tool. Finally, Section 6.11 shows some efficiency experimentation results and Section 6.12 gives a conclusion.

6.2 The Oz Programming Language and the Mozart Programming System

For a better understanding of this chapter, let us first explain some of the characteristics of the Oz language.

6.2.1 The multiparadigm programming language Oz

Oz [50] is a multi-paradigm language that is designed for advanced, concurrent, networked, soft real-time, and reactive applications. It provides the salient features of object-oriented programming, the salient features of functional programming, and the salient features of logic programming and constraint programming including logical variables, constraints and programmable search mechanisms.

The Oz execution model consists of *dataflow threads* observing a shared *store*: A *data flow thread* is a thread that, executing an operation, will suspend it until all operands needed have a well-defined value; i.e., if the statement needs a value that is not yet available, then the thread automatically blocks until it can access that value. The *store* contains unbound and bound logic variables, cells (named explicit state), and procedures. Variables can reference the names of procedures and cells. When a variable is bound, it disappears: all threads that reference it will automatically reference the binding instead. Variables can be bound to any entity, including other variables. The variable and procedure stores are *monotonic*, i.e., information can only be added to them, not changed or removed.

Logic variables An Oz variable is a *single-assignment* variable. Initially, it is introduced with an unknown value, and later it might be assigned a value, in which case the variable becomes *bound*. Once a variable is bound, it cannot be changed. Variables can be bound to variables. Oz is a dynamically typed language. Any variable, if it ever gets a value, will be bound to a value of one of these types

Records, tuples and lists

- A *record* is a compound data structure. It consists of a label followed by a set of pairs `Feature:Field`.

  The following is a record:

  `sol(x:X y:Y z:Z)`

  It has three arguments, and the label `sol`.

  `x,y,z` are features and `X,Y,Z` are fields that can be instantiated or not. An *atom* is a record with no features.

- A *tuple* is a record where we omit the features, reducing it to a compound term.

  So, the following tuple has the same label and fields as the above record:

  `sol(X Y Z)`

  The corresponding record is `sol(1:X 2:Y 3:Z)`.

- A *list* is also an important class of data structures in Oz: a list is either the atom `nil` representing the empty list, or is a tuple using the infix operator `|` and two arguments which are respectively the head and the tail of the list.

  Thus, a list of the first three positive integers is represented as:

  `1|2|3|nil`

  Another convenient special notation for a closed list, i.e. a list with a determined number of elements is: `[1 2 3]`.

These data structures may contain unbound variables. We refer them as *partial values*.

The full Oz language is defined by transforming all statements into a kernel language. Let us briefly define the statements S that will be used in the next sections.

- $X = v$

  v denotes a value (a number, record and procedure).

  The statement `X=v` binds the unbound variable `X` to the integer v, and adds this information to the store. If `X` is already bound to an incompatible value, i.e. to any value different from v, an exception is raised.

- Binding a variable to a value is a special case of the operation named *unification*.

  The unification $term_1 = term_2$ makes the partial values equal if possible by adding bindings in the store.

  Examples:

  `X1 = X2`: if both variables are unbound; one binding between these variables is added to the store.

  `X = f(A)` and `Y = f(25)` : doing `X = Y` binds `A` to `25`

- S_1 S_2

  Inside a thread, a sequence of statements is executed sequentially according to the dataflow: a statement needs that its variables are bound to be executed, the execution of the sequence may be suspended waiting that another thread bounds the variables needed to be bound.

- `local` X `in` S `end`

  All variables are logic variables, declared in an explicit scope defined by the `local` statement.

- A *procedure* is the value of a procedure type.

  The statement `X = proc{$ Y1...  Yn} S end` binds the variable `X` to a new procedure value. That is, it simply declares a new procedure. So, procedures are defined at run-time. The `$` indicates that the procedure value is anonymous, i.e. created without being bound to an identifier.

  A shortcut is

  `proc{`X $Y1..$ Yn`}` S `end`

  - Any statement can be encapsulated into a procedure by putting inside a procedure declaration.
  - A free identifier of a statement is an identifier that is not defined in that statement.

  - A procedure can have external references which are free identifiers in the procedure body. The value of an external reference is its value when the procedure is defined (lexical scoping).

  This characteristics is important, it allows higher order procedures and we often use it in our implementation.

- `fun`$\{F\ X_1\ ...\ X_n\}$ `S E end`

  This function definition is translated into

  `proc`$\{F\ X_1 ...\ X_n\ ?R\ \}\ S\ R = E$ `end`

  `R =` $\{F\ X_1 ...\ X_n\}$ translates the procedure call $\{F\ X_1\ ...\ X_n\ R\}$

- `if` X `then` $S1$ `else` $S2$ `end`

  Conditionals use the keyword `if` and block until the condition variable X is true or false in the variable store.

- The pattern-matching

  ```
  case X of E_1 then S_1
  [] E_2 then S_2
  [] ...
  else S end
  ```

  We often use the case statement with multiple alternatives and complicated conditions. All variables introduced in E_i are implicitly declared, and have a scope stretching over the corresponding S_i.

- `thread` S `end`

  Threads are created explicitly with the `thread` statement.

- $\{$`Wait` $X\}$

  This statement suspends explicitly the current thread until X is determined

6.2.2 Finite Domain Constraint Programming with Oz

The Oz language allows constraint programming over finite domains. In this section, we provide an overview of the Oz constraint programming model, a complete description of which can be found in [49].

Constraints Oz constraint programming uses two kinds of constraints. *Basic constraints* are in the form $x \in D$ where x is a variable and D is a finite subset of the natural numbers, called the *domain* of x. *Non-basic constraints* express relations between variables; a simple example is $x+y \leq z$.

Constraint solving Operationally, computation takes place in a computation space. A *computation space* consists of a constraints store and a set of propagators. The *constraints store* implements a conjunction of basic constraints, which can be dynamically refined by the propagators. A *propagator* is a concurrent computational agent that imposes a non basic constraint by narrowing the domains of the variables involved in the constraint.

Let us assume, for instance, that the constraint store s consists of two basic constraints $x \in \{1, \dots, 6\}$, and $y \in \{1, \dots, 6\}$. Moreover, let us suppose that the propagator pa_1 imposes the constraint $x + 3 = y$. The basic constraints are refined to $x \in \{1, 2, 3\}$ and $y \in \{4, 5, 6\}$ because other values of the domains are not compatible with the constraint $x + 3 = y$. Propagators communicate through the constraint store by shared variables. Consider again our example and let us add another propagator pa_2, that imposes the constraint $y - 2 * x > 1$. Once pa_1 has refined the basic constraints to $x \in \{1, 2, 3\}$ and $y \in \{4, 5, 6\}$, the second propagator ensures that $x \in \{1, 2\}$. Now, pa_1 can propagate again giving $y \in \{4, 5\}$, then pa_2 establishes $x = 1$, and, finally, pa_1 computes $y = 4$. At this moment, the computational space encapsulating s, pa_1 and pa_2 becomes *stable* (i.e., no further constraint propagation is possible). Moreover, one says that this computational space has *succeeded*, which means that the variable assignment $x = 1$ and $y = 4$ is a *solution* to the initial constraint problem. A computational space can also be *failed* if a propagator detects that its associated constraint is inconsistent with a basic constraint.

Variable distribution Constraint propagation is not a complete solution method: It may happen that a set of constraints has a unique solution and that constraint propagation does not find it. Similarly, constraint propagation may be unable to detect that no solution exists. Consider, for instance, the same problem where propagator pa_2 is replaced by propagator pa'_2, that imposes $y - x * x > 1$. After propagation, the computational space gets stable with the following store: $x \in \{1, 2, 3\}$ and $y \in \{4, 5, 6\}$. In such a situation, the computational space is said to be *distributable*, which means that it can be divided into two disjoint computational spaces by splitting the domain of a variable. To do so, we make two copies of the original computational space and we add a propagator that imposes $x = 1$ to the first copy and a propagator that imposes $x \neq 1$ to the second one. Propagators may then wake up in both spaces. The choice of the variable to be distributed and the choice of the value given to this variable is called a *distribution strategy*. The efficiency of constraint solving may heavily depend on the distribution strategy. A popular strategy is the *first-fail* strategy: select a variable with the least number of values. To guarantee a complete solution method, we can simply ensure that all the Oz variables can be distributed.

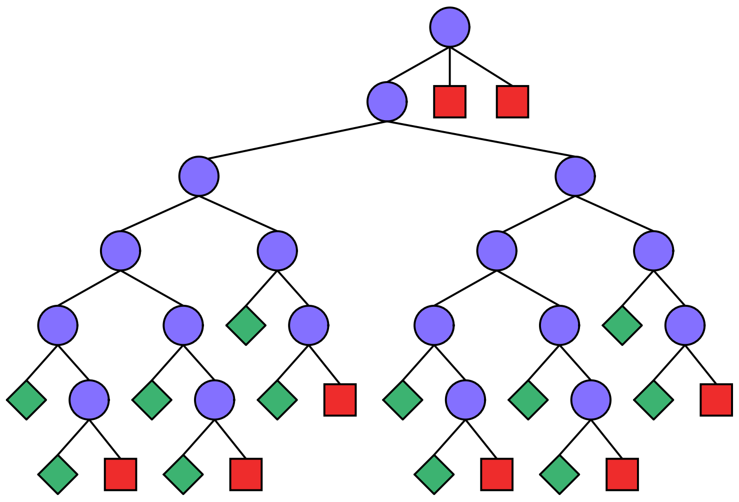


Figure 6.1: An example of search tree

Search trees Search proceeds by distributing the space. Iterating constraint propagation and distribution leads to a tree of spaces, the *search tree*. An example is displayed in Figure 6.1 : each node in the search tree corresponds to a computation space. Leaves correspond to solved spaces (drawn as diamonds) or failed spaces (drawn as boxes), and distributable spaces (drawn as circles). Given the tree search, several strategies are possible to explore it: depth-first or breadth-first exploration. A program that implements exploration is called a *search engine*. The following functionalities are available: search for a single solution, several solutions, or all the solutions. An interactive and visual search is also provided (see Figure 6.1). A resulting search tree must be deterministic, i.e., the distribution of a computation space must be deterministic. One reason is that search engines feature support *recomputation*: the search engine may recompute any node of the search tree instead of cloning it. Concretely, a problem with a large number of variables or propagators or a problem for which the search tree is very deep might use too much memory to be feasible. Recomputation reduces the space requirements for these problems in that it trades space for time.

The script A script for a finite domain problem is a program that can compute one or all solutions of the problem. In Oz, scripts will be run on predefined search engines implementing the propagate and distribute method just described. Separating scripts from the search engines running them is an important abstraction making it possible to design scripts at a very high level. To develop a script for a given problem, we start by designing a model and a distribution strategy. We then obtain an executable script by implementing the model and distribution strategy with the predefined abstractions available in Oz.

In Oz, a script takes the form of a procedure

```
proc {Script Root}
%% declare variables
in
%% post constraints
%% specify distribution strategy
end
```

The procedure declares the variables needed, posts the constraints modelling the problem, and specifies the distribution strategy. The argument `Root` stands for the solutions of the problem to be solved. If the solutions of a problem are given by more than one variable, say `X`, `Y`, and `Z`, we may simply combine these variables into one record by posting a constraint such as
`Root = solution(x:X y:Y z:Z)`.
The procedure `{SearchOne Script ?Solutions}` will run the script `Script` until the first solution is found. If a solution is found, it is returned as the single element of a list; otherwise, the empty list is returned.

6.2.3 The Mozart programming system

Mozart is the system that implements the Oz language. Besides, the Mozart system provides many predefined modules (records that group together a set of related operations). We mention those used in this chapter.

Predefined modules

- The `Dictionary` module contains procedures operating on dictionaries.

  A dictionary is a mapping from simple constants (atoms, names or integers) to partial values. Both the domain and the codomain can be changed.

- The `Finite Domain` module, `FD` is the module of Oz which contains all the useful procedures for finite domain constraint programming. The propagators and the distribution functions are defined there.

- The `List` module contains procedures operating on lists.

- The `Oz Explorer` is a graphical and interactive tool to visualize and analyze search trees.

- The `Search` module describes the different search engines.

6.3 Verification Conditions

To check the correctness of a program with respect to a specification, we use verification conditions. A *verification condition* is a formula that is logically equivalent to propositions of the form $\{P\}S\{Q\}$, where P and Q are assertions and S is a program fragment. All verification conditions have the form $A \Rightarrow B$. Using the weakest precondition method [17], we verify $P \Rightarrow wp(S,Q)$; using the strongest postcondition method [24], we verify $sp(P,S) \Rightarrow Q$.

We have chosen the *sp* approach because it allows us to directly translate the assertion P into a set of constraints equivalent to $sp(P,S)$, i.e., without manipulating the syntactic tree of P. The renaming needed in a assertion transformation with the *sp* method is implicit. Each assertion transformation consists of adding a constraint between two environments of program. When using the *wp* method, the assertion transformation consists in a substitution of a variable with an expression and concretely, we would first transform the syntactic tree of Q into another syntactic tree representing $wp(S,Q)$ before translating this into a set of constraints.

To check that a verification condition $A \Rightarrow B$ is valid, the idea is to find the solutions of a constraint problem equivalent to A & $!B$. If no solution exists, the verification condition is valid. Otherwise, counter-examples to the verification condition are found. It is in this way that the Oz constraint solving system checks whether the verification conditions are valid.

To give a clear idea of the way we translate verification conditions into Oz constraints and how the constraints problems are solved, we propose to detail a part of the verification of the binary search algorithm. This algorithm has been used in Chapter 4 to demonstrate the behaviour of our tool when the algorithm (or the specification) is not correct. It is depicted with its specifications in Figure 4.5. Through this example, we explain a few important matters. Let us check a part of the proposition $\{Inv$ and not $H\}$ $Iter\{Inv\}$, in the case where $a[m] > x$.

$$\{Inv \ \& \ !(g = d | b = false) \ \& \ m = (g+d) \text{ div } 2 \ \& \ a[m] > x\} \ \texttt{d := m} \ \{Inv\}$$

Using the *strongest postcondition* approach (*sp*), we want to prove the following implication:

$$(\exists d_1 : d = m \ \& \ Inv^{d}_{d_1} \ \& \ !(g = d_1 | b) \ \& \ \ m = (g + d_1) div \ 2 \ \& \ \ a[m] > x)$$

$$\Rightarrow Inv$$

The notation $Inv^d_{d_1}$ means that we substitute the new variable d_1 for every free occurrence of the variable d in the formula Inv. Unfolding the formula $Inv^d_{d_1}$, the first part of the implication rewrites to the formula

$$\begin{array}{lll}
(\mathbf{1}) & & (\forall\ i : 1 \leq i \leq n-1 : a_0[i] \leq a_0[i+1]) \\
(\mathbf{2}) & \& & d = m \\
(\mathbf{3}) & \& & a, x \text{ unchanged} \\
(\mathbf{4}) & \& & 1 \leq g \leq d_1 \leq n+1 \\
(\mathbf{5}) & \&\& & (\forall\ i : 1 \leq i < g : a[i] < x) \\
(\mathbf{6}) & \& & (\forall\ i : d_1 \leq i \leq n : a[i] > x) \\
(\mathbf{7}) & \& & b \Rightarrow (\exists\ i : 1 \leq i \leq n : a[i] = x) \\
(\mathbf{8}) & \& & !(g = d_1 | b = true) \\
(\mathbf{9}) & \& & m = (g + d_1)\ div\ 2 \\
(\mathbf{10}) & \& & a[m] > x
\end{array}$$

The second part of the implication is

$$\begin{array}{lll}
(\mathbf{11}) & \& & 1 \leq g \leq d \leq n+1 \\
(\mathbf{12}) & \& & a, x \text{ unchanged} \\
(\mathbf{13}) & \&\& & (\forall\ i : 1 \leq i < g : a[i] < x) \\
(\mathbf{14}) & \& & (\forall\ i : d \leq i \leq n : a[i] > x) \\
(\mathbf{15}) & \& & b \Rightarrow (\exists\ i : 1 \leq i \leq n : a[i] = x)
\end{array}$$

6.4 Translating Verification Conditions in Oz

We now explain how an assertion can be translated into an Oz constraint problem. To obtain the script, we first need to declare the Oz variables and specify their basic constraint. Then, we have to translate the assertions into constraints. We show a direct translation of the verification condition first. Observing the problems encountered with this translation, we motivate our actual method.

6.4.1 Basic Constraints

We define the initial constraint store. For each program variable mentioned in the assertion, according to the program declarations, we generate a basic constraint (one for each element of an array). We recall the declarations of the binary search program:

```
const n = 4;
var x : 0..n ;
tab a : array [1..n] of 0..n+1 ;

var g : 0..maxint ;
var d : 0..maxint ;
var m : 0..maxint ;

var b : boolean ;
```

The corresponding Oz variables declaration are:

```
X :: 0#4
A = {FD.tuple 'tab' 4 0#5}
[G D D1 M]::: 0#FD.sup
B :: 0#1
```

- `X, G, D, D1, M` are the Oz variables corresponding to the program integer variables x, g, d, d_1, m; d and d_1 represent the same program variable but at two different program states.

  `FD.sup` is a constant integer : the largest value of the Oz variables in finite domain constraint programming. Its concrete value is implementation dependent: 134 217 726.

- The array a is defined as a tuple with the following basic constraints: `A.1::0#5, A.2::0#5, A.3::0#5, A.4:0#5`.

- The Boolean variable b is represented by a simple Oz variable for which the domain is $\{0, 1\}$ where 0 corresponds to false and 1 corresponds to true.

6.4.2 Naive Translation

We manually translate the verification condition. In fact, not all subassertions can be directly (i.e., statically) translated into Oz constraints. For instance, the translation of assertion **(5)** depends on the value of the variable g. When needed values are not known statically, translation must be done dynamically (when these needed variables become bound to a single value). Thus, for simplicity, we limit ourselves to assertions whose translation is direct.

$$\begin{array}{lll}
(\mathbf{1}) & & (\forall\ i : 1 \leq i \leq n-1 : a[i] \leq a[i+1]) \\
(\mathbf{2}) & \& & d = m \\
(\mathbf{4}) & \& & 1 \leq g \leq d_1 \leq n+1 \\
(\mathbf{8}) & \& & !(g = d_1 | b = true) \\
(\mathbf{9}) & \& & m = (g + d_1)\ div\ 2 \\
(\mathbf{11}) & \& & !(1 \leq g \leq d \leq n+1)
\end{array}$$

The corresponding Oz script that we can naturally write is depicted in Figure 6.2: To every subassertion corresponds a set of propagators; a number expresses these correspondences.

- We mainly use relational propagators: `=:`, `>`, and `=<:`.

- The relation **(9)** involves a non trivial arithmetic expression:

  to translate it, we use an auxiliary Oz variable `X1`.

- To translate **(8)**, we first give another form to this assertion:

  $g \neq d_1$ & $b = false$.

- We translate the quantified assertion **(1)** into a conjunction before translating it into propagators.

- Assertion **(11)** is first translated into $1 > g \mid g > d \mid d > 5$,

  then to specify the disjunction, we use a `choice` statement.

- `Sol=sol(a:a(A.1 A.2 A.3 A.4) g:G d1:D1 d:D m:M)`

  stands for the solution of the constraint problem; it is explained in Section 6.2.2.

- `{FD.distribute ff [A.1 A.2 A.3 A.4 G D1 D M X1]}`

  is the call of the predefined distribution method using the *first-fail* strategy on the given list of variables; we do not focus on this matter in this section.

We first notice that to translate our assertion into constraints, we have normalized the subexpressions to post the adapted constraints. Besides, using a `choice` statement seems straightforward to specify a disjunction. In fact the choice statement specifies the alternatives with which the space is to be distributed (see Section 6.2.2). It means that we increase the number of spaces, which will not be easily manageable if we begin to have "nested" disjunctions. To avoid these problems, the key choice of our implementation is to encapsulate each assertion into a Boolean variable.

6.4.3 Using Reified Constraints

Reified constraints are of the form $c \leftrightarrow b$ where c is a non basic constraint and b is a Boolean (0/1) variable. If $b = 1$, the reified constraint is equivalent to c. If $b = 0$, it is equivalent to $\neg c$. The main reasons to use reified constraints are:

- Any formula of our interpreted logic can be translated into a single conjunction of reified constraints, i.e., into a single constraint problem. So, we avoid to increase the number of computational spaces to manage disjunctions.

  Let us consider a formula of the form $A|B$. It can be translated to the conjunction of reified constraints

  $(b = b_A|b_B)$ & c_1 & $\ldots$ & c_n & c'_1 & $\ldots$ & $c'_{n'}$

```
declare
proc {VC Sol}
   A B G D1 D M B X1
in
   A = {FD.tuple tab 4 1#5}     %% declaration variables
   B::0#1
   [G D1 D M]:::0 # FD.sup
   [X1]:::0# FD.sup

   D  =: M          (2)           %% post constraints
   1  =<: G         (4)
   G  =<: D1        (4)
   D1 =<: 5         (4)

   X1 =: G + D1     (9)
   {FD.divI X1 2 M} (9)

   G \=: D1         (8)
   B =: 0           (8)

   A.1 =<: A.2      (1)
   A.2 =<: A.3      (1)
   A.3 =<: A.4      (1)

   choice          (11)
      G >: D       (11)
   []
      G <: 1       (11)
   []
      D >: 5       (11)
   end

   Sol = sol(a : b(A.1 A.2 A.3 A.4) g : G d1 : D1 d : D m : M)

   {FD.distribute ff [A.1 A.2 A.3 A.4 G D1 D M X1]}
   %% specify distribution strategy
end
{Explorer.one VC}
end
```

Figure 6.2: The naive translation of a verification condition

where the c_i are the reified constraints translating A and b_A is the Boolean variable associated to A (similarly for B).

- The translation of our assertions into constraints is more systematic: it does not require any normalization to be translated into constraints.
- The encapsulation of each subexpression, as tiny as it can be, of an assertion will allow us to systematically determine the violated part of this assertion.

Translation of the example Let us translate each subexpression of the example into a list of reified constraints

$$(\mathbf{4})\begin{cases} b41 = (1 \leq g) \\ b42 = (g \leq d_1) \\ b43 = (d_1 \leq n+1) \\ b4 = b41 \ \& \ b42 \ \& \ b43 \end{cases} \qquad (\mathbf{2})\{\ b2 = (d = m)$$

$$(\mathbf{9})\begin{cases} z = g + d_1 \\ y = z \ div \ 2 \\ b8 = (m = y) \end{cases} \qquad (\mathbf{8})\begin{cases} b91 = (g = d_1) \\ b92 = (b = 0) \\ b93 = (b91 \mid b92) \\ b9 =! \ b93 \end{cases}$$

$$(\mathbf{1})\begin{cases} ba_1 = (a_0[1] \leq a_0[2]) \\ ba_2 = (a_0[2] \leq a_0[3]) \\ \ldots \\ ba_{n-1} = (a_0[n-1] \leq a_0[n]) \\ b1 = (ba_1 \ \& \ ba_2 \ \& \ \ldots \ \& \ ba_{n-1}) \end{cases} \qquad (\mathbf{11})\begin{cases} b111 = (1 \leq g) \\ b112 = (g \leq d) \\ b113 = (d_1 \leq n+1) \\ b114 = b111 \ \& \ b112 \ \& \ b113 \\ b11 =! \ b114 \end{cases}$$

With $n = 4$, the list of corresponding propagators is

```
(4)
B41 = 1 =<: G
B42 = G =<: D1
B43 = D1 =<: 5
B44 = {FD.conj B41 B42}
B4 = {FD.conj B44 B43}

(8)
X1 =: G + D1
X2 = {FD.divI X1 2} % X2 =: X1 / 2
B8 = (M =: X2)

(2)
B2 = (D =: M)
```

```
(9)
B91 = (G =: D1)
B92 = (B =: 0)
B93 = {FD.disj B91 B92}
B9 = {FD.nega B93}

(1)
BA1 = A.1 =<: A.2
BA2 = A.2 =<: A.2
BA3 = A.3 =<: A.4
BA4 = {FD.conj BA1 BA1}
B1 = {FD.conj BA3 BA4}

(11)
B111 = (1 =:< G)
B112 = (G =:< D)
B113 = (D =<: N + 1)
B114 = {FD.conj B111 B112}
B11 = {FD.conj B114 B113}

(1) & (2) & (4) & (8) & (9)
R1 = {FD.conj B4 B8}
R2 = {FD.conj B2 B9}
R12 = {FD.conj R1 R2}
R13 = {FD.conj B1 R12}
```

These examples show

- reified propagators (to encaspulate relations)
- 0/1 propagators (to propagate conjunction and disjunction and negation)

6.5 Solving the Constraints

The subset of assertions that we have translated in the previous section corresponds to the following verification condition:

$$\left\{\begin{array}{ll} & (\forall\ i : 1 \leq i \leq n-1 : a[i] \leq a[i+1]) \\ \& & d = m \\ \& & 1 \leq g \leq d \leq n+1 \\ \& & !(g = d | b = true) \\ \& & m = (g+d) \text{ div } 2 \\ & \end{array}\right\}$$

```
d := m
```

$$\{(1 \leq g \leq d \leq n+1)\ \}$$

Suppose the user writes a wrong statement: he writes `d := m-1` instead of `d := m`. This is a scenario that we have considered in Chapter 4. Let us show how the tool concretely finds the counter-examples and how it is able to give a precise feedback about the violated part of the invariant.

Let us recall what we have to solve. We want to verify the correctness of $\{P\}\ S\ \{Q\}$; we translated it into $sp(P,S) \Rightarrow Q$ where $sp(P,S)$ corresponds to **(1)** & **(2)** & **(4)** & **(8)** & **(9)** which is encapsulated into `R13` and Q corresponds to **(11)** and is encapsulated into `B11`. Notice a modification in **(2)** since we consider another statement:

```
(2)
B2 = (D =: M-1)
```

To prove the correctness or to find a counter-example: we impose

```
R13 = 1  % sp(P,S) is true
B11 = 0  % Q is false
```

The propagators try to reduce the domains of the variables. Unfortunately, constraint propagation is not a complete solution method: we may need to distribute variables. Let us have a look at the example and let us distribute manually, in an intuitive way.

If `R13=1`, propagation is able to determine that `R12=1` and `R3=1`.
If `R12=1` then `R1=1` and `R2=1` is computed by propagation and so propagation imposes `B4=1, B8=1, B2=1, B9=1, B1=1, BA5=1`.

- `B4=1`: propagation determines `B1=1`, `B2=1` and `B3=1`.

  Propagation refines the domains of `G` and `D1`:

  `G::1#5 D1::1#5`

- `B8=1`: propagation reduces the domains of `M`, `X1` and `X2`:

  `X1::2#10 X2::1#5 M::1#5`

- `B2=1` impose `D=:M-1`, the domain of `D` is refined:

  `D::0#4`

- `B9=1`: propagators impose `B93=0, B91=0, B91=0` and

  `B` is determined: `B=0`.

- `BA5=1`: propagation gives `BA3=1, BA4=1, BA4=1, BA1=1, BA2=1`

  but the propagators cannot refine the domains of the elements of `A`.

- `B11=0`: we should need to distribute on `B113=0` and `B114=0`.

  But in parallel, propagation has determined `B111`, `B113`, `B112`, and `B114`

  - `B111=1`: because `(1=:<G)` is true
  - `B113=1`: because `(D=<:5)` is true
  - `B112=0`: because `B111=1` and `B113=1` and `B11=0`
    `B112` reifies `(G<=:D)`, so
    `D<:G` is imposed, but no variable domain is refined.
    The domains of `G` and `D` stay unchanged(*):
    `G::1#5 D::0#4`

We have simulated the way propagators refine the variables domains. The chronology of the propagators actions is arbitrary, since actually they are concurrent agents. Besides they may "wake up" several times: if the domain of `G` and `D` is changed in (*), all the other propagators wake up to try to refine again. Well, we come back to our example and, now, the computational space gets stable and all the Boolean variables are determined. We have the following store:

```
[A.1 A.2 A.3 A.4]:::0#5  B=0
G::1#5 D::0#4 D1::1#5
```

We use the *first-fail* distribution strategy on this set of Oz variables:

```
{FD.distribute ff [G D A.1 A.2 A.3 A.4 X1 X2 B]}
```

`D` and `G` have the same smallest domain size (> 1); we distribute for example, on `G`:

- `G=1` is added to the store.

  The values of `D, D1 M, X1, X2` are determined:

  `D=0 M=1 X2=1`

  Thanks to the propagators corresponding to **(8)**

  `D1::2#5 X1::2#3`

  And, finally, after several refining from the propagators of **(8)**, `X1=3 D1=2`

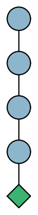
  We finally distribute on the elements of `A`.

- `G/=:1` ...

The first solution is

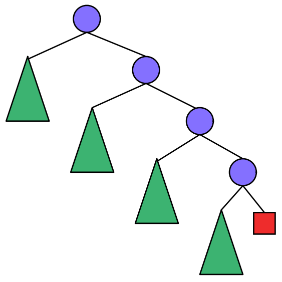
```
a(1 1 1 1) b = 0 d = 0 d1 = 2 g = 1 m = 1
```

On the search tree:



The violated part in Q is encapsulated in `B114` and, more precisely, is encapsulated in `B112`. Indeed, the search consists of finding the violated subassertions in Q. Q is a conjunction of subassertions $A1$ & $A2$ &...& An; $\neg Q$ becomes the disjunction $\neg A1 \mid \neg A2 \mid ... \mid \neg An$. To determine which subassertion is violated, we have to determine which subassertions $\neg Ai$ are true, i.e, which Ai are encapsulated in $false$ (i.e., 0).

The following figure shows the whole search tree (large green triangles contain sets of solutions). The red square corresponds to the computational space where `G=5`, this fails:



Our example does not require distributing on the Boolean variables. If we had considered the complete verification condition introduced in Section 6.3, we would have needed to distribute on the Boolean variables. Figure 6.3 shows the whole search tree containing all the counter-examples of $\{Inv$ and not $H\}$ $Iter\{Inv\}$ with the wrong statement `d:= m-1`. The set of small green diamonds corresponds to the set of counter-examples for which the violated part of the invariant is $g <= d$, the large green triangle corresponds to the set of counter-examples for which the violated part of the invariant is $(\forall\ i : d \leq i \leq n : a[i] > x)$. All the blue circles above the large triangles (red and green) are distributable spaces on which a Boolean variable has been distributed.

The choice of the variable we want to distribute has an impact on efficiency. Ideally, the programmer manually chooses by experimentation the

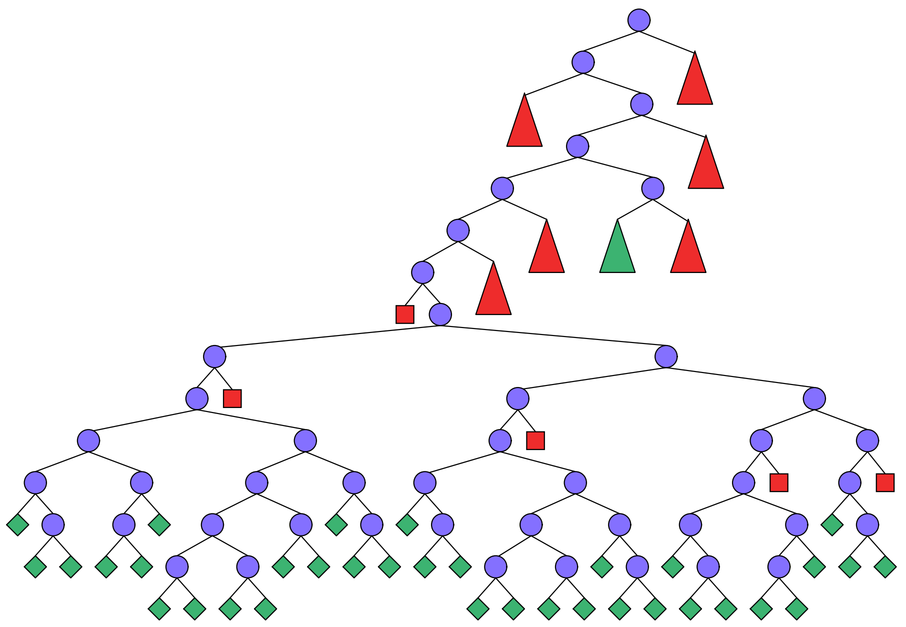


Figure 6.3: the search tree of $\{Inv$ and not $H\}$ $Iter\{Inv\}$ with the wrong statement `d:= m-1`

best strategy of distribution for the particular problem he solves. The difficulty for us is that we do not choose a strategy for a particular problem, but for an automatically generated set of problems. According to the problem, the best strategy may be different. After experimentation, a good strategy is to first distribute the Boolean variables encapsulating subexpressions:

- The Boolean Oz variables encapsulating subexpressions of an expression can be distributed only if the Boolean encapsulating the expression is determined.
- The integer Oz variables of an arithmetic expression can be distributed only when the Boolean encapsulating the relation involving this arithmetic expression is determined.
- Among the variables that can be distributed, we choose the variable to be distributed using the *first-fail* strategy

To perform it, we use a data structure containing all the Oz variables involved in the set of propagators translating the assertion. This structure gives priority to variables that we want to distribute first, according to our distribution strategy. For example, the data structure of the subassertion **(4)** is the following:

```
DS4 = [ B4 #[ B44#[B41#[G] B42#[G D1] ] B43#[D1] ]
```

This data structure is a tree such that for each subtree, the root is a Boolean variable. It means, for instance, that `G` can be distributed only if `B42` or `B41` is determined.
For **(8)**, the data structure is:

```
DS8 = [B8#[ M X2 X1 G G1] ]
```

The solution of this constraint satisfaction problem is complete because all the involved Oz variables are distributable.

6.6 Automating the Translation

6.6.1 Use of Dictionaries

In the manual version, our Oz variables are identified with a capital letter and by convention, we mention the program variables by small letters. Concretely, in the automatic translations, we use dictionaries that maintain the correspondence between the program variables identifiers and their corresponding Oz variables. To prove the correctness of $\{P\}\ S\ \{Q\}$ (P and Q may be in relation with the precondition (Pre)), we use the following dictionaries:

- one dictionary that corresponds to the state of the data at the Pre program point, we name it `Mp0`,
- one dictionary to represent P, the state of the program variables before the execution of S,
- one dictionary to represent Q, the state of the program variables after the iteration,
- intermediate dictionaries for the intermediate program points between P and Q,
- a new dictionary for every instantiation of a free variable (we explain it at the end of this section).

6.6.2 Pattern-matching on the Syntactic Tree

Using pattern matching, we automatically translate the syntactic tree of each assertion into the corresponding set of reified constraints: `Mp` is the dictionary corresponding to the set of states of the program point where we want to evaluate `Assert`. `Mp0` is the dictionary corresponding to the precondition states. The reader is advised to refer to Section 5.3: an assertion is evaluated on two environments. While generating the propagators, we construct the data structure `DS`.

```
proc{PropB Assert Mp Mp0 DS B}

case Assert of
  ident(X) then {Dictionary.get Mp X}=: B
              DS = [B]
[]
  >(X Y) then V1 V2
         in
           [V1 V2]:::0#FD.sup
           {PropA X Mp Mp0 DS1 V1} % X =: V1
           {PropA Y Mp Mp0 DS2 V2} % Y =: V2
           DS = [B#[DS1 DS2]]
           B = : V1 > V2
[]
   and(A1 A2) then B1 B2
              in
                [B1 B2]::0#1
                {PropB A1 Mp Mp0 DS1 B1} % B1 =: A1
                {PropB A2 Mp Mp0 DS2 B1} % B2 =: A2
                DS = [B#[DS1 DS2]]
                {FD.conj B1 B2 B} % B = B1 and B2
[]
   ...
end
```

{Dictionary.get Mp X} takes in Mp the Oz variable corresponding to the programming variable represented by X.

We have a similar procedure for arithmetic and generalised arithmetic expressions (defined in Section 5.3).

```
proc{PropA Gaexpr Mp Mp0 DS V}

case Gaexpr of
  ident(X) then {Dictionary.get Mp X}=: V
              DS = [V]
[]
  indent0(X) then {Dictionary.get Mp0 X}=: V
                DS = [V]
[]
  +(X Y) then V1 V2 DS1 DS2
         in
           [V1 V2]:::0#FD.sup
           {PropA X Mp Mp0 DS1 V1} % X =: V1
           {PropB Y Mp Mp0 DS2 V2} % Y =: V2
           DS = {List.append DS1 DS2}
           V = : V1 + V2
[]
   ...
```

```
end
```

Again, we notice that all the variables involved in the constraint satisfaction problem appear in the data structure `DS`; no matter the priority rules of this structure, all the variables can be distributed. The solution of the generated constraint satisfaction problem is complete.

6.6.3 Translating Quantified Assertions

We come back to our example of verification condition in Section 6.3 and we look at the following quantified assertion where n is fixed ($n = 4$):

$$(\mathbf{1})(\forall i : 1 \leq i < n : a_0[i] \leq a_0[i+1])$$

The list of reified constraints is:

$$(\mathbf{1}) \begin{cases} ba_1 = (a_0[1] \leq a_0[2]) \\ ba_2 = (a_0[2] \leq a_0[3]) \\ ... \\ ba_{n-1} = (a_0[n-1] \leq a_0[n]) \\ b1 = (ba_1 \ \& \ ba_2 \ \& \ \dots \ \& \ ba_{n-1}) \end{cases}$$

For each i, we have a Boolean variable that encapsulates the constraint $a[i] \leq a[i+1]$. Then, we have a list of conjunctions to be forced to 1. Functionnal programming is very well adapted to the translation of quantified assertions/expressions into constraints (with the function Map and Fold) if we consider the index list:

Figure 6.4 depicts the function that generates a reification of the constraint

$$\forall x : i \leq x \leq j : p(x)$$

where `I` and `J` are integers and `P` is such that `B_X = {P X}` is a reified constraint of $p(x)$. The data structure `DS` is the list of the data structures of each $p(x)$.

To translate **(1)**, we instantiate (first manually) the parameter `P` with a function with a parameter `I`, defining the reified propagator corresponding to $a_0[i] \leq a_0[i+1]$:

```
  P = fun{$ I} BI = A0.I =<: A0.(I+1)
               DSI = [BI#[A0.I A0.(I+1)]]
               BI#DSI
      end
in
  B1 = {ForEach 1 3 P DS}
```

We write `A0` to mention a_0. In the automatisation, `A0` is an anonymous tuple standing in `Mp0` for which the key is the atom `a`:

```
   A0 = {Dictionary.get Mp0 a}
```

```
proc {ForEach I J P DS B}
      Dom = {List.number I J 1} (1)
in
      DS = {Map Dom P}          (2)
      B = {FoldL Rs fun{$ X Y#_} {FD.conj X Y} end 1}  (3)
end
```

(1)Dom =[I, I+1,...,J]
(2)DS = [P I, P I+1, ... P J] = $[B_I\#DS_I, B_{I+1}\#DS_{I+1}, ...B_I\#DS_J]$
(3)B = B_I & B_{I+1} & ...& B_J

Figure 6.4: Generating propagators for $\forall x : i \leq x \leq j : p(x)$

Figure 6.5 shows the automatic instantiation of the the paramater `P`. In our example, (without speaking about `DS`).

- {`PropA A1 V1 Mp Mp0 DS1`} generates `V1=:1` and so determines `V1`,

  {`PropA A2 V2 Mp Mp0 DS2`} generates `V2=:4` and so determines `V2`,

- each `BK = `{`PropB P Mpl Mp0 DSK`} generates $a_0[i] \leq a_0[i+1]$.

  To perform it, it uses a new dictionary which is a clone of the current dictionary, containing one more variable whose key is i.

6.7 Interleaving Constraint Translation with Constraint Solving

The previous discussion ignores a major difficulty in our method, which is that the number and the form of some constraints may depend on the value of one or several variables. To overcome this difficulty, we interleave constraint generation and constraint solving. Consider the following assertion:

$$(\mathbf{5})(\forall\ i : 1 \leq i < g : a[i] < x)$$

We would like to translate it into the following reified constraint whereas the variable g is not determined:

$$(\mathbf{5}) \begin{cases} bb1 = (a[1] < x) \\ bb2 = (a[2] < x) \\ \dots \\ b5 = (bb1\ \&\ bb2\ \&\ \dots) \end{cases}$$

We would like to execute the `ForEach` method which is defined in Figure 6.4. But, there is a *precondition* to execute the procedure of `ForEach`: the values of variables `I` and `J` must be determined. To ensure this precondition,

```
proc{PropB Assert Mp Mp0 DS B}

case Assert of

...
[]
  'forall'((ident(X) '<=' A1 '<=' A2) Assert) then
     P=fun {$ K} BK Mpl DSK in
          {Dictionary.clone Mp Mpl}
           K = {Dictionary.put Mpl X}
           BK = {PropB P Mpl Mp0 DSK}
           BK#DSK
        end
     V1 V2
  in
     {Propa A1 Mp Mp0 DS1 V1}
     {Propa A2 Mp Mp0 DS2 V2}
     B = {ForEach V1 V2 P DS}
[]
 ...
```

Figure 6.5: Translating the formula $(\forall\ x : a1 \leq x \leq a2 : p(x))$

we can encapuslate the `ForEach` call in a thread and add explicit `WAIT` statements for the actual parameters corresponding to `I` and `J` before any call to the procedure `ForEach`. It is important to put this procedure call into a thread because it is a generator of propagators that are concurrent agents and that must be executed independently from each other. For example, if we do not use any thread to translate a conjunction of two quantified expressions, the second one cannot be translated before the first one because the procedure calls are executed sequencially (see Section 6.2).

```
{PropA A1 Mp Mp0 DS1 V1} % propagators V1 =: A1
{PropA A2 Mp Mp0 DS2 V1} % propagators V2 =: A2

local P = fun{$ I} BI = A.I <: X
                   DSI =[BI#[A.I X]]
                   BI#DSI
           end
in
 thread {Wait V1}{Wait V2}
      B = {ForEach V1 V2 P DS}
 end
end
```

where `A1` and `A2` correspond resp. to 1 and $g-1$. In a more general way, they correspond to extended numerical expressions such as defined in Section 5.3

```
proc{PropB Assert Mp Mp0 DS B}

case Assert of

...
[]
  'forall'((ident(X) '<=' A1 '<=' A2) Assert) then

      P= fun {$ K} BK Mpl DSK in
           {Dictionary.clone Mp Mpl}
           K = {Dictionary.put Mpl X}
           BK = {PropB P Mpl Mp0 DSK}
           BK#DSK
         end
      V1 DS1 V2 DS2 DS
   in
      DS = [[DS1 DS2]# L]
      {Propa A1 Mp Mp0 DS1 V1}
      {Propa A2 Mp MP0 DS2 V2}
      thread {Wait V1} {Wait V2}
            Bool = {ForEach V1 V2 P L}
      end
[]
```

Figure 6.6: Translating the formula $(\forall\ x : a1 \leq x \leq a2 : p(x))$

($\mathit{eaexpr} \in \mathit{Eaexpr}$). Notice that `P` is automatically generated, but here, we define it manually for better readability. Figure 6.6 contains the adapted version of 6.5.

To translate $(\forall i : 1 \leq i \leq g-1 : a[i] < x)$ into constraints, we need to wait that g is determined. The number of propagators and the new Oz variables added in the constraint store depend on this value `G`. Concretely, some constraints must be solved before generating other propagators. But, without distribution, this variable `G` may never be determined and the solving of this problem may result in a deadlock. So, although every variable may possibly need to be distributed, variables that must be determined to allow reified constraint generation must be distributed first. The definition of `DS` in Figure 6.6 is so justified: the variables contained in `DS1` and `DS2` must be distributed first. The word "first" gives a priority notion but in fact, the Oz variables corresponding to the quantified assertion are not yet generated, `L` is undefined until the procedure `ForEach` is called.

The call to procedure `ForEach` will not be delayed forever: First the variable `DS`, which is part of the partially data structure used for distribution, is further instantiated with two new variables `DS1` and `DS2`; then procedure `Propa` is executed to create two threads that will both evaluate `V1` and `V2` and instantiate `DS1` and `DS2` with the variables to be distributed in `A1` and `A2` (i.e., the Oz variables corresponding to the variables occurring in the expressions $a1$ and $a2$ of the assertion). As soon as `V1` and `V2` are determined, the `ForEach` function can generate its propagators and instantiate the data structure.

We come back to the example (**5**)($\forall\ i : 1 \leq i < g : a[i] < x$):

```
DS5=[ B#[ [[1] [G]]#[_]]]
```

As soon as `G` is determined, (we choose for example `G = 4`), the function `ForEach` generates the propagators of $a[1] < x$ & $a[2] < x$ & $a[3] < x$, and the structure of the variables to be distributed immediately becomes (see Figure 6.4 for the construction of this part of the data structure):

```
B#[ [1 3]#[B1#[A.1 X] B2#[A.2 X] B3#[A.3 X]]]
```

Next the distribution strategy will choose `B1`,`B2` or `B3` if propagation cannot compute them and the process continues, alterning propagation and distribution.
The translation of (**6**)($\forall i : d \leq i \leq n : a[i] > x$) into constraints is similar: we need to wait until `D` is determined.

Another example of translation into constraints that needs interleaving contraint generation and constraint solving is when we need to consult an element of an array:

$$(\mathbf{10})\ a[m] > x$$

is translated into

$$(\mathbf{10}) \left\{ \begin{array}{l} a[m] = v \\ b10 = (v > x) \end{array} \right.$$

To refer $a[m]$ in a Oz constraint, we need to wait until the index m is determined.

```
    T = {Dictionary.get Mp A}    % T is the tuple corresponding to A
in
    {PropA Y Mp Mp0 DS I}        % Propagator I =: Y
    thread {Wait I} V =: T.I end % Propagator V =: T[I]
```

`Y` represents an extended numerical expression *eaexpr* that may involve several variables possibly to be distributed.

We have a similar automation for the existential quantifier and for the generalised arithmetic expressions involving quantifiers such as `sum, max, min, #`.

6.8 Distribution Heuristics - Completeness of our Method

To guarantee a complete solution for our constraint satisfaction problems, all the involved Oz variables must be distributable, the resulting search tree must be deterministic (even if in each computation space propagators run in a non deterministic way) and also, no deadlock should append.

Through the examples of Section 6.7, we have seen that propagators cannot be posted at the same time, as well as the data structure `DS` that cannot be completely built at the same time (statically). The propagators are generated dynamically while the data structure is built. So, there are many reasons to doubt about the completeness of our method. Let us explain the role of the data structure `DS` to guarantee a complete solution; it is a tree for three reasons:

- The first reason is that the resulting search tree must be deterministic. Using this data structure of the dynamically generated variables, the distribution of a computation space is deterministic.

  Indeed, if we add the generated Oz variables at the end of a flat structure (a list), the distribution in a computation space may be non deterministic since the generation of the Oz variables are concurrent processes. The search tree will not be unique, and if the search engine recomputes a node from the tree root, the solution of the problem will be random.

  Our data structure is deterministic, as well as our distribution strategy (the way we select the variable to be distributed).

- Next, we have seen that the set of propagators and the new Oz variables added to a constraint store may depend on the value of one or several variables. The data structure allows us to easily select the part of the tree which is to be considered at each distribution step.

  Each time we distribute, we use an adapted first-fail heuristic on the list of the selected variables. It means that we look for one of the selected variables with the smallest domain and we distribute on this variable (this choice is deterministic).

- Finally a minor reason that does not mention completeness but efficiency: the structure gives priorities on the variables to be distributed, which allows us to improve efficiency of the search.

In fact, using this data structure, we have a complete method. In each stable computation space, (see 6.2.2) all the Oz variables that are created with a propagator generation are systematically in the structure `DS` and so, are distributable. A current constraint satisfaction problem always has a complete solution. Of course, it does not necessarily correspond to the whole problem we want to solve since some propagators may not be already translated. The `WAIT` statements involved in the translation of some assertions may be a source of deadlock. To avoid it, the Oz variables involved in the `WAIT` statements are systematically added to the current computation space and are added in the data structure. Since any current computation space has a complete solution, we guarantee that the execution of the running `WAIT` statements will terminate.

6.9 Final Improvements

6.9.1 Detecting Run-time Errors and Incorrect Assertions

Up to now, we have never mentioned that we may translate badly defined expressions. A way to manage them is catching the errors while the script is running. For instance, if we consult an element of an array which is not defined because of an out-of-bounds error, we stop the search and give a feedback concerning the error. In our examples of assertion, it may occur when we propagate `V=:T.I` or when we run the function `ForEach` or `ForOne` with a too large range.

The Boolean operator && can be translated in an operational way. For the following example

$$\begin{array}{lll} & (\mathbf{4}) & 1 \le g \le d_1 \le n+1 \\ \&\& & (\mathbf{5}) & (\forall\ i : 1 \le i < g : a[i] < x) \\ \& & (\mathbf{6}) & (\forall\ i : d_1 \le i \le n : a[i] > x) \end{array}$$

we can first solve the constraint problem related to assertion **(4)**, and then generate the propagators corresponding to assertions**(5)** and **(6)**. Since our example is well defined, no error can be generated.

The simplified generic method corresponding to this translation is the following:

```
fun{AndLr P1 P2}
   R = {P1}
in
   thread
     if R == 1 then
        {P2}
     else
```

```
        0 end
end
```

This function generates the propagators corresponding to $assert_1$ && $assert_2$ P1 and P2 are such that R1={P1} is a reified constraint of $assert_1$ and R2={P2} is a reified constraint of $assert_2$.

Unfortunately, this way to manage errors does not correspond to the semantics described in Chapter 5. Indeed, the propagators are concurrent computational agents and so, according to the order in which the propagators run to narrow the domains of the variables involved in the constraints, an error can be detected or not (the result is non determinist). The following example shows two propagators that will fail the space, but also a propagator defining a badly defined expression: A.0.

```
A={FD.tuple a 4 1#4}
I::0#4
B::0#FD.sup
G <:4
G >: 4
thread {Wait I}
D =: A.I
end
```

With respect to our semantics, each assertion should be encapsulated in a variable with three values: $true$, $false$ and $error$. Since Oz constraints cannot be reified in a variable with three values, our implementation considers ordered pairs of Boolean variables (B, Er): B has to be considered as in the previous Section ($true$ or $false$), Er encapsulates the fact that the assertion is badly defined or not. Of course, if Er=1, the value (determined or not) of B has no meaning. Let us have a look at some of the subexpressions of our example:

$$(\mathbf{10})\ a[m] > x$$

The expression $a[m]$ is badly defined if m is out-of-bounds of a, i.e, if $m < 1 \mid m > n$ (assuming $a[1..n]$), the identifier x cannot be badly defined. Using again reified constraints, we have:

$$(\mathbf{10}) \begin{cases} e1 = (m < 1) \\ e2 = (m > n) \\ e10 = e1|e2 \end{cases}$$

$$(\mathbf{5})(\forall\ i : 1 \leq i < g : a[i] < x)$$

This assertion is badly defined if the range is badly defined or if there exists i in $[1..g]$ such that $a[i]$ is badly defined.

$$(\mathbf{5})\begin{cases} 1 < g \\ 1 < i \\ e1 = (i < 1) \\ e2 = (i > n) \\ e5 = e1|e2 \end{cases}$$

Through the examples, we have observed that the constraints generated when we want to impose an assertion to be true or false do not correspond to the same set of constraints that we generate when we want to impose that the assertion is badly defined. It means that the set of propagators that we generate depends on the value of the variable `Er`.
With this choice of implementation, the Boolean operator && has now an implementation that totally matches with our semantics:

```
 proc{AndLr P1 P2 B Error}
      B1 B2 Er1 Er2 :::0#1
in
      {P1 B1 Er1}
      {P2 B2 Er2}
      B = {FD.conj B1 B2}
      Er = {FD.disj Er1 {FD.conj B1 Er2}}
   end
```

This function generates the propagators corresponding to $assert_1$ && $assert_2$ `P1` and `P2` are such that `(Er1 B1) = {P1}` is a reified constraint of $assert_1$ and `(Er2 B2) = {P2}` is a reified constraint of $assert_2$. With this implementation, the propagators of $assert_1$ and $assert_2$ are generated at the same time.

Similarly, for an arithmetic expression, we have an ordered pair `(V,Er)` where `V` corresponds to the integer value of the expression if `Er = 0`, i.e., the arithmetic expression is well defined, otherwise, `V` has no meaning and `Er = 1`. Let us have a look at the generation of the propagators:

```
proc{PropB Assert Mp Mp0 DS B Er}

case Assert of

  ident(X) then {Dictionary.get Mp X}=: B
             DS = [B]
             Er = 0
[]
  >(A1 A2) then
```

```
      {PropA A1 Mp Mp0 DS1 V1 Er1 } % V1 =: A1 or A1 = error (Er1=1)
      {PropA A2 Mp Mp0 DS2 V2 Er2 } % V2 =: A2 or A2 = error (Er2=1)
      Er =  {FD.disj Er1 Er2}       % error propagation
      thread if Er == 0 then        % if A1 and A2 are well defined
                B = (V1>:V2)        % B = A1 > A2
             end
      end
      DS = [[B Er]#[DS1 DS2]]
[]
...
```

In both cases (arithmetic expressions or assertions), the set of propagators that we generate depend on the value of the variable `Er`. For each Boolean subexpression the data structure containing the variable to be distributed has the following form:

`DS= [Er,B]#[the involved Oz variables of the expression]`

As soon as `Er` is determined, the generation of the propagators of the expression runs and initialises in parallel the corresponding parts of the data structure `DS`. For arithmetic expression we have a similar reasoning.

Choosing the depth first search strategy, we can first focus on the search of a badly defined assertion (the first branch of the tree) and then, searching for a counter-example violating the verification condition we consider. This case corresponds to the one we have considered in the previous sections (error problems are ignored).

6.9.2 Dealing with Negative Expressions

A disadvantage of Oz is that it does not manage negative values.

Example Suppose that a student chooses a negative variant for the binary search algorithm, for example,

`Variant: g-d`

Our tool should discover a counter-example showing that the variant is negative. To perform it, we cannot naively translate the arithmetic expressions (*aexpr*). Let us have a look at the following set of propagators:

```
...
Var :: 0#FD.sup
G <: D1
Var =: G - D1
B = (var <: 0)
```

This set of constraints does not have any solution (since `Var::0#FD.sup` and the propagators will reduce the domain of `Var` to empty) and so no counter-example is found. Using this method, the tool would be unsound.

With respect to our semantics of Chapter 5, we consider that each *aexpr* can be encapsulated into an integer value in $[0..maxint] = Val$ or *neg* or *error*. To support it with Oz, we use a similar implementation as for the variable with three values of the previous section: we use a tuple `(V,Neg,Er)` where `V::0#FD.sup` and `Neg::0#1` and `Er::0#1`. `Er = 1` means that the *aexpr* is badly defined, the variable `V` and `neg` have no meaning. Otherwise, `Neg =1` means that the *aexpr* is negative, the variable `V` has no meaning; otherwise, `V` represents the positive value of the *aexpr*.

To give an idea of the way it works, let us consider the following expression:

Examples

$$(\mathbf{11})\ g - d < 3$$

The principle consists in encapsulating in the variable `Neg` the fact that $g - d < 0$: we write `Neg =G<:D` and only when `Neg=0`, we consider the propagator `V=:G-D`. So the example $g - d < 3$ is translated into:

```
...
V :: 0#FD.sup
Neg = G <: D
thread if Neg = 0 then
          V =: G - D
        end
end
B = (V <: 3)
1 = {FD.disj Neg B}   %G -D < 0 or (G-D = V) < 3
```

Let us have a look at the automatic generation of propagators of an arithmetic expression. Of course, a simple identifier cannot be negative. However, an addition of two expressions is considered to be negative if both expressions are negative, i.e., `Neg1` and `neg2` are equal to 1. If only one of the subexpression is negative, we consider that the result is $\top$ taking abstract interpretation notions; as explained in Chapter 5.2, we consider it as an error. In the last case, when both variables are positive, we post the propagator of the addition. We exactly follow the semantics described in Table 5.6. For a subtraction, we need to encapsulate the comparison between the two expressions to know if the result is negative or not, as explained in Table 5.4.

```
 proc{PropA Gaexpr Mp Mp0 DS V Er Neg}
  case Gaexpr of
     ident(X) then
       B = {Dictionary.get Mp X}
       Er = 0
       Neg = 0
```

```
      DS = [V]
  []
    '+'(X Y) then V1 V2 DS1 DS2 Er1 Er2 Neg1 Neg2 in
      {PropA X Mp Mp0 DS1 V1 Er1 Neg1}
      {PropA Y Mp Mp0 DS2 V2 Er2 Neg2}
      DS = [[Er Neg]#[DS1 DS2]]
      Er = {FD.disj {FD.disj Er1 Er2} {FD.exor Neg1 Neg2}}
      Neg = {FD.conj Neg1 Neg2}
      {FD.plus V1 V2 V}
[]
    '-'(X Y) then V1 V2 DS1 DS Er1 Er2 Er3 Neg1 Neg2 Neg3 in
      {PropA X Mp Mp0 DS1 V1 Er1 Neg1}
      {PropA Y Mp Mp0 DS2 V2 Er2 Neg2}
      DS = [[Error Neg Neg3]#[DS1 DS2]]
      Er = {FD.disj {FD.disj Er1 Er2} Neg2}
      Neg = {FD.disj {FD.conj Neg1 {FD.nega Neg2}} Neg3}
      thread if Er == 0 then
               thread if Neg3 == 0 then {FD.minus V1 V2 V}
                      else Neg3 = (V1 <: V2)
                      end
               end
             end
      end
...
```

Let us have a look at the propagators generation of an assertion. One can observe that the translation of a comparison follows exactly the semantics described in Table 5.9. If both expressions are negative, the result is undetermined (considered as an error), otherwise the case can be handled.

```
proc{PropB Assert Mp Mp0 DS B Er}

case Assert of
     ident(X) then {Dictionary.get Mp X}=: B
             DS= [B]
             Er = 0
[]
     '>'(X Y) then
         X1 X2 DS1 DS2 Er1 Er2 Neg1 Neg2
     in
         {PropA X Mp Mp0 DS1 V1 Er1 Neg1}
         {PropA Y Mp Mp0 DS2 V2 Er2 Neg2}
         DS = [[Er1 Neg1]#DS1 [Er2 Neg2]#DS2]
         Error = {FD.disj {FD.disj Er1 Er2} {FD.conj Neg1 Neg2}}
         thread if Error == 0 then
                   if Neg1 == 1 andthen Neg2 == 0 then Bool = 0
                   elseif Neg2 == 1 andthen Neg1 == 0 then Bool = 1
                   else Bool = (V1>:V2)
                   end
```

```
                end
            end
...
```

The following example shows that we can handle some cases with a negative bound in the range of a quantified variable:

$$(\mathbf{5})(\forall i : 1 \leq i \leq g - 1 : a[i] < x)$$

If $g-1$ is evaluated to *neg*, i.e., `Neg2 = G<:1` entails `Neg2=1`, then the range is empty and we do not need to call the `ForEach` method, the quantified assertion can be evaluated to *true*. So, when we call the procedure `ForEach`, we suppose that `Neg2 = 0`.

Thanks to these final improvements, our method is sound, i.e., no error is forgotten. In fact, the principles of finite domain constraint programming make it a sound method, but without a particular handling of negatives and badly defined expressions, the automatic translation of our programs into constraint satisfaction problems is not correct. With these improvements, the semantics described in Chapter 5 is totally satisfied.

6.10 Architecture of MPVS

Figure 6.7 provides a view of the architecture. The structure is simple. Concretely, subproblems files are translated into syntactic trees. From syntactic trees, a procedure generates a script corresponding to the verification condition to be checked; this script is executed by a search engine to look for counter-examples. If no counter-example is found, the user receives a feedback testifying the total correctness of the Hoare proposition he checks. If a counter-example is discovered, all the feedback information is in the counter-example. Let us insist on the fact that we do not need any postprocessor to analyse the result of the search engine.

Let us focus on each part of the diagram of Figure 6.7:

- The input consists of the declaration, the code and the specification of the problem to be checked. In the simplest case, it corresponds to one file. When a decomposition into subproblems is needed, several files must be provided: the main problem file, plus the files of the called subproblems from which the tool only needs declarations, pre and postconditions.
- Parsing of the input files is the first task of MPVS. The tool parses the main problem and automatically/recursively parses the files of the called subproblems.

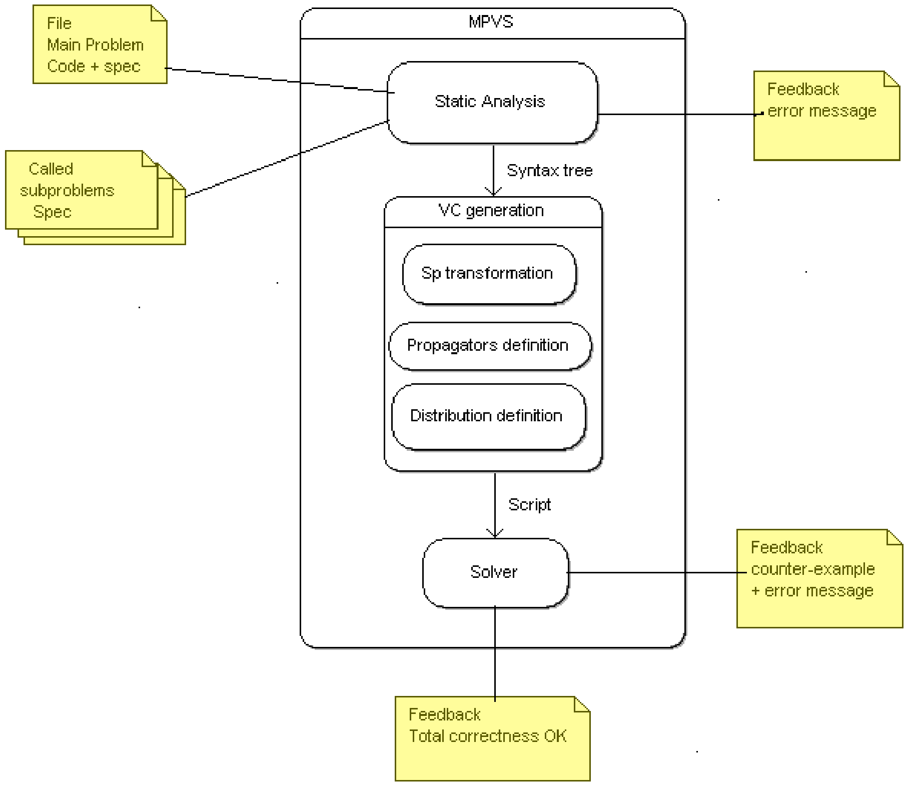


Figure 6.7: The MPVS architecture

This task includes type checking according to the declaration. The declaration of all the subproblems are global and must be coherent with each other, following the semantics described in Section 5.2.

- This syntactic analysis may return an error message (including the position in the file) if a problem has occurred.

  If no error is detected, it returns a set of syntactic trees; each one is identified by a subproblem name (i.e, the file name).

- According to these syntactic trees and the Hoare proposition we want to check, let $\{P\}\ S\ \{Q\}$ given a precondition Pre, a procedure generates the script; a script is itself a procedure which contains:

  - The declarations of Oz variables on finite domain: one for each symbolic variable of the verification condition:
    - Statically from Pre, P and S, we generate the dictionaries mapping the program variables to the generated Oz variables, denoted by `Mp0`, `Mp` and `Mp1`. The renaming is implicit thanks to these dictionaries.
  - The propagators corresponding to the translation of the associated verification condition into constraints;
    - to search for a counter-example testifying that Pre is badly defined,
    - to search for a counter-example testifying that P is badly defined,
    - to look for a runtime error imposing the constraints corresponding to S badly defined,
    - adding some constraints between dictionaries `Mp` and `Mp1`, according to the assignment S, we make implicitly the strongest postcondition method:
      - to look for a counter-example testifying that Q is badly defined,
      - to look for a counter-example violating Q.
  - The description of the distribution strategy with its data structure containing the variables to be distributed.

    As explained in Section 6.7, the data structure `DS` is dynamically built during the generation of the propagators; `MyDistribution` procedure, partially shown in Appendix B.3.2, is an adaptation of the first-fail strategy.

Details about the organization of the procedures for the generation of the script are given in Appendix B.3.2.

- The resulting script runs on a predefined search engine, exactly as explained in Section 6.2.2.
- The result of the search engine is an empty list if there is no counter-example. Otherwise, the result is a single element list containing a record of the following form

  ```
  Sol = sol(Mp0 Mp Mp1 Violated InvolvedS Feedback Error)
  ```

  where `Mp0`, `Mp` and `Mp1` are the dictionaries corresponding to three environments of the program (see Section 6.6), it corresponds to the three columns appearing in each counter-example of Chapter 4.

  `Error` allows us to know if the source of the counter-example is a badly defined expression. If not, i.e., if `Error` is equal to 0, it means that the problem stands in the partial correctness of the Hoare proposition.

  `Violated` is a record that represents the violated or badly defined subassertion (including positions in the file). This information allows us to underline the involved expressions in the editor.

  `InvolvedS` is a record that represents the sequence of statements involved in the problem; it is also underlined in the file.

  `Feedback` is a record that expresses, when `Error = 1`, the kind of error detected if we have a badly defined expression: a division by zero, an out of bound error, or even a conjunction of different kinds of error.

6.11 Efficiency Experiments

We provide execution times corresponding to the complete verification of a set of simple programs.
`Square` computes the square of x using the relation $(x+1)^2 = x^2 + 2x + 1$. `Divide` computes the quotient and the reminder of dividing x by y. `Power` computes x^y in $\log(y)$ time (indian exponentiation). `SSearch` is a sequential search of x in a. `VSearch` is a binary search algorithm to find the minimum value of a "valley" a, where a valley consists of a decreasing sequence followed by an increasing sequence. `BSearch` is a binary search algorithm (presented in [20]). `CSmall` finds the number of values smaller than x in a valley. `Common` finds the number of common values of two strictly sorted arrays a and b. `NPerm` is an algorithm to find the next permutation in lexicographic order (also presented in [20]). It uses three subproblems. `NComb` is a similar algorithm to find the next combination of n numbers out of $1, \ldots, m$; it also uses three subproblems.
In Figure 6.8, the second and third columns give the restrictions that were put on the variable domains to check the programs. The fourth and fifth columns provide the number of variables and the number of single statements

Algo	Program				Verification		
	Data+domains	const	v	i	Oz v	prop	times
`Square`	$x \in [0..4]$		4	7	96	137	31ms
`Divide`	$x \in [0..16]$ $y \in [0..4]$		4	5	98	422	94ms
`Power`	$x \in [0..4]$ $y \in [0..4]$		5	10	152	2150	656ms
`SSearch`	$x \in [0..1]$ $a[1..n] \in [0..n]^n$	$n \leq 4$	4	5	797	906	90ms
`VSearch`	$a[1..n] \in [0..n]^n$	$n \leq 4$	5	8	47648	39181	741ms
`BSearch`	$a[1..n] \in [0..n]^n$ $x \in [0..1]$	$n \leq 4$	5	11	21359	16459	900ms
`CSmall`	$a[1..n] \in [0..n]^n$ $x \in [0..1]$	$n \leq 4$	5	8	148463	107501	1090ms
`Common`	$a[1..n] \in [0..n]^n$ $b[1..m] \in [0..n]^m$	$n \leq 4$ $m \leq 4$	5	11	89951	70952	2794ms
`ISort` `SPIsort`	$a[1..n]$	$n \leq 4$	4	9	3476 2754	81152 43422	570ms 600ms

`NPermut`
`SP1`
`SP2`
`SP3`
	$a[1..n] \in [0..n]^n$	$n : 1..5$	6	19	206799
					12979
					2284
					16334
						168840
						9664
						112
						1148
							4496ms
							200ms
							430ms
							701ms

`Ncomb`
 	
`CSP1`
`CSP2`
`CSP3`
	$a[1..n] \in [0..m]^n$	$n \leq 4$
		$m = 6$
			6	16	11030
					
					2531
					1138
					913
						13929
						
						2024
						926
						707
							350ms
							
							140ms
							100ms
							47ms

Figure 6.8: Efficiency results

in the programs. The sixth and seventh columns indicate how many Oz variables (i.e., basic constraints) and propagators (i.e., non basic constraints) are generated by the verification system. Finally, the last column contains the execution times.

6.12 Conclusion

In this chapter, we have explained the automatic translation of verification conditions into constraints problems. Through our examples of implementation, we have shown that Oz is a powerful language which is very well adapted to generate our constraint problems. Oz multiparadigm brings a better language flexibility, which makes the development shorter. The use of reified constraints allows a systematic translation of the assertions into propagators. The functional programming principles of Oz and the use of data structures like dictionaries and tuples facilitates this systematic translation. Using threads, we can interleave constraint propagation and constraint solving, and also dynamically define our data structures. We never mention the graphical interface that has been implemented in Oz as well. It gives the advantage of allowing interaction between the search engine, the data structures and the graphical interface. The student can easily keep on searching counter-examples after he receives the feedback from the previous one.

Using another finite domain constraint programming system to implement our tool would probably be possible, but then, some code parts would probably be more complex. For instance, cloning a dictionary in Oz is made by Mozart. It clones the data stucture together with the variables and the variable bindings, which is not necessary straightforward with some other languages. However, Oz is a very powerful language, but it is also very difficult to learn and to deeply understand. Besides, it does not handle negative values.

We could have built our verification tool using SMV but this option seems to require interoperability between several systems what we avoid by using Oz: translating the algorithm with its specification into scripts (one for each array size, one for each Hoare proposition), checking the scripts with SMV, and then exploiting the feedbacks given by SMV to provide a clear feedback to the student. A comparison between the two approaches has been published in [19] and is described in [22].

We could also have used a SAT Solver [42]. We do not have more deeply investigated SAT Solver techniques. CBMC [11] is an example of tool that follows this approach: it unrolls a given ANSI-C program up to a given bound on each loop and recursion depth. Then, it translates the resulting transition relation to propositional logic, assuming the given (finite) type of each variable (e.g. an integer is represented by a 32-bit vector). Next, it adds the negation of user-defined assertions to the formula. Finally, it sends the resulting formula to a SAT solver. But this tool does not handle quantified assertions and precisely, we think that:

- The translation of these assertions with a non determined range would be heavy (the translation should be static).
- Encoding our assertions would be more complex than the translation into constraints.
- Similarly, the output of the SAT Solver must be translated into a clear feedback to the user. Besides, we think that, using a SAT Solver, our assertion structure is lost and that the feedback analysis becomes more difficult.

> We have seen in this chapter that keeping the structure of our assertions and our code is important to pinpoint the errors which can help the student to understand his reasoning errors.

Notice that our objective was to write a pedagogical tool, and not to use the tool giving the most efficient results. The tool certainly gave us very rapidly good results. However, it could have been interesting to study more deeply the heuristic distributions to try to choose automatically a set of representative counter-examples.

Part III

Experimentation

Chapter 7

Using MPVS in a Programming Course

In this thesis, we propose to use a pedagogical software to stimulate the students to deeply understand and to use the programming method based on the invariant and decomposition into subproblems. After the evaluation of some existing tools, we conclude that it is difficult to get an existing software that is appropriate enough for our pedagogical objectives. Using a technique of program verification based on finite domain constraint Programming, we have completely implemented our own verification tool; this tool gives optimistic results in the sense that it is fully automatic, it only requires the abilities we want to teach and no extra skills; and it is able to give precise feedback to the student in moderate time.

In this context, it would be interesting to analyse the impact of such a tool on the behaviour, the motivation and the understanding of the students who learn this programming method. We have the opportunity to teach the "Program Conception Methods" course [39] to the computer science students in the third year of their studies. Among these students we note that there are three different orientations (engineers, computer science bachelors, graduate). The goal of this course [1] is to teach the declarative method to construct programs. Students learn how to construct programs arguing that it is correct, a priori, by using a logical reasoning. Concretely, a part of the course was the teaching of the methodology described in this thesis, i.e., the teaching of the programming method based on loop invariants and decomposition into subproblems.

During the last two years, we have experimented our tool to support the teaching of this programming method. The first year, we used the tool

[1]The professor is Baudouin Le Charlier in the Computing science department INGI of UCL

during the first four weeks of the course. The second year, the tool was used after some theoretical lessons about this method and after some exercises on paper. At each time, we strongly insisted on the fact that the students have to think about a solution on paper before encoding their solution in the tool.

In practice, MPVS is available on the students' machines of the computing science department; besides, students can easily install it on their own machine. All exercises are carefully prepared, as detailed later in this chapter, and they are transmitted to the students via the web.

In this chapter, we first develop what we want to evaluate. Then, we explain the progress of the course and labs for both years. We present the two different approaches of using our tool and we provide evaluations. Finally, we conclude with positive points as well as some less positive observations that we try to explain.

7.1 What Do we want to Evaluate?

We are convinced that an appropriate tool verifying the correctness of programs written with the method based on the invariant and the decomposition into subproblems should help and motivate the student. Through this learning, students should learn many other important notions, they should learn to think a program in a declarative way, they should learn to **decompose a problem** into subproblems, they should understand the role of a **complete specification**, they should learn to be **rigorous** and improve their **abstraction level**. Let us give a list of questions that we must look at concerning the goals reached by the tool.

- Does the tool enforce the use of the method?
- Does the tool help to understand the method?
- Does the tool help the students to integrate the level of abstraction needed to use this tool?
- Does the tool motivate the students to learn and to write formal specifications?
- Does the tool help the students to understand what a complete specification means?
- Does the tool enforce rigor?
- Does the tool give more interest in formal proofs and formal methods in general?

- Does the tool charm the students with the witchery of the algorithm correctness proofs?
- Does the tool convince students to write programs with this method?

Thanks to our observations during the lab sessions and the evaluation reports provided by the students, we discuss the answers to these questions.

7.2 A First Experimentation

7.2.1 The Course Organisation

We made our first experimentation during the first weeks of the academic year 2005-2006 (October and November). In this experimentation, the tool is used in the first part of the course "Program Conception Methods" [39]. The progress of the course is the following. First, we make a demonstration of the tool to the students, then, during four weeks, exercises with increasing difficulties are given to them. From a practical point of view, a lab session is organised every week, as well as a theoretical lesson to review deeply the exercises made during the lab session. Besides, students are invited to hand out each week a report with some of their solutions, their self-evaluation (which results from the use of the tool), and manual proofs combining symbolic execution of the statements and logical reasoning [2].

In this section, we detail the four lab sessions by providing some exercises submitted to the students (the solution to these exercises are in Appendix B.5); one will observe that they are carefully prepared: specifications are precise, formally or informally; the declaration of the data is given, and, for each variable, we have chosen a judicious finite domain. Let us detail the four lab sessions.

Lab 1

We begin with an introduction to the tool. The teacher illustrates the tool with examples fully specified and coded. Students have the possibility to change some statements or assertions and can understand the counter-examples. Students get familiar with the tool and the language. Then, Java versions of the algorithm are given with their informal specifications (the sequential search example is presented in Figure 7.1); students must translate them into our tool and check the correctness. Next, we provide exercises considering algorithms with the same pre and postconditions but with another loop invariant. Figure 7.2 displays the exercise of the sequential search exactly as it is given to the students; they must fill in the code. In

[2]Examples of manual proofs are given in Chapter 2.

this exercise, the invariant does not involve any Boolean variable contrary to the Java version given in Figure 7.1.

These exercises may already highlight the students on the method principles. At this lab session, many students did not understand the role of the invariant, and filled in the code without looking at the invariant. We observed students that were not able to understand by themselves the counter-examples.

Lab 2

The second lab session involves a bit more complicated exercises completely specified (with a given loop invariant), and without any code. The students are asked to write the code based on these specifications. One of these exercises is the Belgian flag algorithm which is presented in Figure 7.3.

Lab 3

The third lab session includes exercises specified (without any given loop invariant). The students have to choose an invariant and then fill in the statements. One of the exercises is the algorithm finding the number of values common to two strictly sorted arrays (see Figure 7.4).

Confronted with these exercises, the students have difficulties to formalise their invariant, they should understand the efficiency of a reasoning on picture before formalising; they also notice that it is not simple to have a complete invariant and why it needs to be complete. Notice that since the students are free to choose their invariant, they also have to choose (to declare) their auxiliary variables.

Lab 4

Finally, during the fourth session, the students have to construct more complex algorithms requiring a decomposition into subproblems. Specifications are often given informally but the decomposition is completely given to the students; the student just needs to implement each subproblem of the decomposition.

Figures 7.5, 7.6 and 7.7 display the three subproblems composing the algorithm computing the number of occurrences of a value in a sorted array in $\log(n)$ time using binary searches. This exercise shows the importance of writing precise pre and postconditions in subproblems, even informally.

```
class rechSeqDG
{
static boolean find(int x, int[] a)
/* Pre      : a != null
   Post     : the array a is unchanged
   Result   : true if an element of a is equals to x ;
              false otherwise
*/
{
  int i = a.length ;
  boolean present = false ;

  /* Invariant :
     ---------
  - 0 <= i <= a.length ;
  - present = true if and only if the subarray a[i .. a.length-1]
                   contains an element equals to x.
     Variant : i
     -------     */

  while (i != 0 & !present)
  { i-- ; present = (a[i] == x) ; }

  return present ;
}

public static void main(String[] args)
{
  System.out.println(find(0, new int[]{})) ;
  System.out.println(find(0, new int[]{0})) ;
  System.out.println(find(0, new int[]{1})) ;
  System.out.println(find(0, new int[]{0, 1})) ;
  System.out.println(find(0, new int[]{1, 1})) ;
  System.out.println(find(0, new int[]{1, 0})) ;
  System.out.println(find(0, new int[]{0, 0})) ;
}
}
```

Figure 7.1: A Java version of the sequential search with its informal specification and invariant

```
Data: const n <= 5 ;
      const minv = 0 ;
      const maxv = 2 ;
      tab   a : array [1..n] of minv .. maxv ;
      var   x : minv .. maxv ;

Auxiliary_variables:
      var   i : 0 .. maxint ;

Result_variables:
      var   present : boolean ;

Precondition: initialised(x) & initialised(1, n : a)

Postcondition: unchanged(x) & unchanged(1, n : a) &&
      (present <=> (exist i : 1 <= i <= n : a[i] = x))

Invariant: unchanged(x) & unchanged(1, n : a) & 0 <= i & i <= n
           && (forall j : 1 <= j <= i : a[j] != x)

Init:

Iter:

Clot:

Halting_condition:

Variant:
```

Figure 7.2: The sequential search exercise: the code is to fill in

```
Data: const n <= 5 ;
  const noir = 7 ;
  const jaune = 8 ;
  const rouge = 9 ;
  tab a : array[1 .. n] of noir .. rouge ;

Auxiliary_variables:
  var in : 0 .. maxint ;
  var ij : 0 .. maxint ;
  var ir : 0 .. maxint ;
  var x : noir .. rouge ;

Precondition: initialised(1, n : a)

Postcondition: permut(a, a_0, 1, n, 1, n) &
  (exist i : 0 <= i <= n :
     (exist j : i <= j <= n :
        (forall kn : 1     <= kn <= i : a[kn] = noir)
      & (forall kj : i + 1 <= kj <= j : a[kj] = jaune)
      & (forall kr : j + 1 <= kr <= n : a[kr] = rouge)
  ))

Invariant:  0 <= in & in <= ij & ij <= ir & ir <= n &&
  permut(a, a_0, 1, ir, 1, ir) & unchanged(ir + 1, n : a) &
  (forall kn : 1 <= kn <= in : a[kn] = noir) &
  (forall kj : in + 1 <= kj <= ij : a[kj] = jaune) &
  (forall kr : ij + 1 <= kr <= ir : a[kr] = rouge)

Init:

Iter:

Clot:

Halting_condition:

Variant:
```

Figure 7.3: The Belgian flag exercise: the code is to fill in

```
Data:
  const m <= 3 ;
  const n <= 3 ;
  const minv = 45 ;
  const maxv = 51 ;
  tab a : array [1 .. m] of minv .. maxv ;
  tab b : array [1 .. n] of minv .. maxv ;

Auxiliary_variables:

Result_variables:
  var k : 0 .. maxint ;

Precondition:
  (forall i : 1 <= i <= m - 1 : a[i] < a[i + 1]) &
  (forall i : 1 <= i <= n - 1 : b[i] < b[i + 1])

Postcondition:
  unchanged(1, m : a) & unchanged(1, n : b) &
  k = (# v : minv <= v <= maxv :
             (exist i : 1 <= i <= m : a[i] = v) &
             (exist j : 1 <= j <= n : b[j] = v)
       )

Invariant:

Init:

Iter:

Clot:

Halting_condition:

Variant:
```

Figure 7.4: The number of values common to two strictly sorted arrays: the invariant and the code are to fill in

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:
  var   g : 0 .. n ;
  var   d : 0 .. n ;

Result_variables:
  var  nv : 0 .. n ; // number of occurrences of v in a.

Precondition:
  // a is sorted (not necessarily strictly) ;
  // v est initialised.

Postcondition:
  // a and v are unchanged ;
  // nv is the number of occurrences of v in a.

Instr:
```

Figure 7.5: The number of occurrences of the value `v` in the array `a` in $\log(n)$ time: the specification and the code are to fill in

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:

Result_variables:
  var   d : 0 .. n ;

Precondition:
  // see main subproblem

Postcondition:
  // a et v are unchanged ;
  // d is the index of the last occurrence of v in a
  // (if it exists) ; otherwise, d is the index of the last element
  // in a that is smaller than v (if there is one) ;
  // otherwise ... (what is logical)


Invariant:

Init:

Iter:

Clot:

Halting_condition:

Variant:
```

Figure 7.6: Subproblem SP1 using a binary search to compute the number of occurrences of `v` in `a` in $\log(n)$ time: specifications, invariant and code are to fill in

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:

Result_variables:
  var   g : 0 .. n ;

Precondition: initialised(v) &
  // see main subproblem

Postcondition:
  // g is the index of the last element of a
  // small than v (if it exists) ;
  // otherwise ... (what is logical)

Invariant:

Init:

Iter:

Clot:

Halting_condition:

Variant:
```

Figure 7.7: Subproblem SP2 using a binary search to compute the number of occurrences of `v` in `a` in $\log(n)$ time: specifications, invariant and code are to fill in

7.2.2 Evaluation

In a general way, we have noticed a good motivation among the students: the number of exercises handed out was way over our expectations and with much higher quality than in the previous years, when students had to solve exercises with pen and paper.

Another positive experience is the direct dialog between the tool and the students. Is seems very interesting, as it directly gives a feedback to the students and motivates them to improve their program (and specifications), and to keep trying to do it better, while the classical approach involves only one evaluation by the teacher. More specifically, the misunderstanding of the method appears for many students in the first lab session. They do not focus on the invariant to build the code of the sequential search; with the tool, by analysing the counter-examples, students finally understand problems that they could even not imagine when they were not using the tool.

Nevertheless, students keep bad habits, they often ignore the error messages and the counter-example: *it doesn't work, why?* and despite our efforts, students have many difficulties to take a sheet of paper and a pen to think over, before encoding their solution in the tool. As a consequence, the tool may be used with an approach similar to the process of "fixing" bugs in programs by trial and error.

7.3 A Second Experimentation

We made another experimentation in the middle of the academic year 2006-2007 (March and April). In this second experimentation, we have proceeded in a different way; instead of starting with the tool, we started with more theory and exercises on paper. So, the tool is used when students have (at least in theory) a deeper understanding of the methodology. Let us explain in more detail the organisation of the course before and during the tool experimentation.

7.3.1 The Course Organisation

The first lesson is devoted to discuss some questions mentioned in the paper [40]: we explain the meaning of a logical reasoning, of an operational reasoning and the ambiguity of the notion of formal reasoning. Then, during two lessons, the teacher constructs the algorithm to find the next combination in the lexicographic order. It is a nice and complex example where the decomposition into subproblems and good specifications make the elaboration easier. In parallel, during lab sessions, we focus on both formal and informal specifications (Most teachers consider that students are able to formalise, unfortunately many students have difficulties to formalise, even in the third year of their studies), and we insist on the role of complete specifications; we make some exercises of decomposing into subproblems, we construct some algorithms, and prove them manually using symbolic execution. Afterwards, a first project is given to the students : to construct in Java an efficient algorithm finding the greatest value corresponding to the sum of a non empty rectangular subarray of a two-dimensional array $a[1..m][1..n]$; the goal is not to have a trivial algorithm with a complexity $\mathcal{O}(m^3n^3)$, but with a complexity $\mathcal{O}(m^2n)$ or possibly even better. The main operational steps of the algorithm are introduced by the teacher. We do not detail this exercise because a very similar project is completely explained later in this section. For this exercise, only 11 students out of 54 succeeded completely, most of the other students had problems of capacity overflow. A description of the tests and the students results are in Appendix B.4.

Then follows a four week training period with the tool and a project in group to reinforce what was learned in the course. The four lab sessions are similar to those given the previous year and the project is detailed below.

7.3.2 The Project

In this section, we describe the project. Then, by showing the way the algorithm works, we divise a decomposition into subproblem, we discuss a solution and, thanks to the students's questions and the reports that students have handed out, we evaluate the impact of the tool on what the students have learned.

7.3.2.1 Description of the project

The goal of the project is to construct and verify with our tool a program specified as follows:

```
Data: const n : ...;
      const m : ...;
      const minv = ... ;
      const maxv = ... ;
      tab   a : array [1..m][1..n] of minv .. maxv ;
      var   S : 0 .. m * n * maxv;

Result_variables:
      var present : boolean ;

Precondition:
initialised(1,n*m:a) & initialised(S)
```

```
Postcondition:
unchanged(a)  & unchanged(S) &&
   (present <=> (exist i : 1 <= i <= m : (exist j : i <= j <= m :
                  (exist k : 1 <= k <= n : (exist l : k <= l <= n :
                      S = (sum ij : i <= ij <= j :
                         (sum kl : k <= kl <= l : a[ij][kl]))
               )))))
```

The specification says that `present` is set to `true` if there exists a non empty subarray of the two-dimensional array `a` for which the sum of the values of all of its elements is equal to `S`; the value `false` is assigned otherwise.

For example, `present` is set to `true` for `n = 3`, `m = 3`, `S = 103` and if the array `a` is the following:

$$\texttt{a :}\quad \begin{array}{r|c|c|c|} \multicolumn{1}{r}{} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{n} \\ \cline{2-4} 1 & 12 & 23 & 56 \\ \cline{2-4} & 6 & \mathbf{61} & \mathbf{17} \\ \cline{2-4} m & 97 & \mathbf{23} & \mathbf{2} \\ \cline{2-4} \end{array}$$

because we have a subarray (represented in bold) that has a sum equal to 103.

Notice that the size of the data is not fixed, the students can choose for preliminary tests; the teacher will provide domains later. Since the automatic tool does not manage two-dimensional arrays, the student needs to simulate the two-dimensional array `a[1..m][1..n]` by an array `a[1..n*m]`. The variable `a[i][k]` is represented by `a[n*(i-1)+k]`.

7.3.2.2 Motivation of the project

In the previous project, the students were asked to follow rigorously the structured programming method that they had learned. The test results, displayed in Appendix B.4, show that most students had a wrong algorithm. By giving the students the opportunity to insert their algorithms into the tool, the students could have a precise feedback about the correctness of their algorithms following the structured programming method. Besides, the use of this method could be enforced for students that did not correctly follow it.

Since the tool does not manage negative values, the algorithm finding the greatest value corresponding to the sum of a non empty rectangular subarray of a two-dimensional array cannot be easily adapted. It is the reason why the project has been a little modified. The difficulties of both projects are nevertheless equivalent.

7.3.2.3 Steps of the project

The steps to be followed are precisely provided by the teacher:

Reasoning with a pen and a sheet of paper Together, students must decompose the problem into subproblems; these must be described in a precise way.

Then, each student chooses one subproblem to solve: he finds the formal pre/post conditions of his subproblem and, if necessary, he completes/modifies the decomposition into subproblems with his co-workers.

Next, each student manually constructs individually a solution of his problem and makes a correctness proof using symbolic execution.

Using the tool Then, the student can encode his algorithm in the tool; he may discover errors and, in this case, he analyses the counter-examples, reasons again on his algorithm, modifies it and concludes about his learning.

Handing out a report Each group must hand out a project report; individually, each student has to give a report concerning his subproblem solving. He is asked to mention the difficulties he has met and to explain the role of the automatic tool in the resolution of the problem.

7.3.2.4 The decomposition into subproblems

To have in mind the way this algorithm works, we have a look on the example:

- At the beginning, the analysed part of the subarray is empty, no subarray of sum S is found.

$$\begin{array}{cc|c|c|c|}
 & & 1 & & \mathtt{n} \\ \cline{3-5}
\mathtt{a}: & 1 & 12 & 23 & 56 \\ \cline{3-5}
 & & 6 & 61 & 17 \\ \cline{3-5}
 & \mathtt{m} & 97 & 23 & 2 \\ \cline{3-5}
\end{array}$$

- Then, we consider the first line of the array and we search for a subarray of sum S. The search corresponds to find a subarray of sum S in a one-dimensional array; if we find it, we stop the algorithm; in the example, there is no subarray of sum S in line 1.

$$\begin{array}{ll|c|c|c|} & & 1 & & n \\ \cline{3-5} a: & 1 & 12 & 23 & 56 \\ \cline{3-5} & & 6 & 61 & 17 \\ \cline{3-5} & m & 97 & 23 & 2 \\ \cline{3-5} \end{array}$$

- Next, we analyse all the subarrays for which the last line is on line 2, i.e: we search for a subarray of sum `S` in line 2; then, we search for a subarray of height 2, for which the sum is `S`, on lines 1 and 2.

  In the first two lines of the array `a`, no subarray of sum `S` is found.

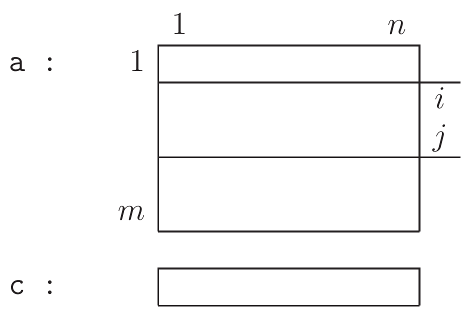
$$\begin{array}{ll|c|c|c|} & & 1 & & n \\ \cline{3-5} a: & 1 & 12 & 23 & 56 \\ \cline{3-5} & & 6 & 61 & 17 \\ \cline{3-5} & m & 97 & 23 & 2 \\ \cline{3-5} \end{array}$$

- Finally, we analyse all the subarrays for which the last line is on line 3.

  On lines 2 and 3, we find a subarray for which the sum is `S`. We can stop the algorithm and set `present` to `true`.

$$\begin{array}{ll|c|c|c|} & & 1 & & n \\ \cline{3-5} a: & 1 & 12 & 23 & 56 \\ \cline{3-5} & & 6 & \mathbf{61} & \mathbf{17} \\ \cline{3-5} & m & 97 & \mathbf{23} & \mathbf{2} \\ \cline{3-5} \end{array}$$

Defining the subproblem SP1 As the step between each state of the simulation is not simple, we define a subproblem `SP1` : the goal of `SP1` is to update the Boolean *present* if there is a subarray of sum S in the subarray $a[1..j][1..n]$ for which the last line is on line j. Operationally, `SP1` searches for a one-dimensional subarray of sum S in line j; if it does not find it, it searches for a subarray of sum S and height 2 in the subarray $a[j-1..j][1..n]$; etc. The subproblem stops when it has found a subarray of sum S or when it has analysed all the possibilities in the subarray $a[1..j][1..n]$: we have an index i to determine the maximal height of the subarrays that we have analysed:



Decomposing SP1 The iteration of this problem is not simple: it searches for a subarray of sum S for which the first line is on line i and the last line is on line j. To perform it, the technique consists in summing the elements of each column of the array $a[i..j][1..n]$ in an auxiliary array $c[1..n]$. Then, the goal reduces to search for a subarray of sum S in the one-dimensional array c. For efficiency reasons, we keep at each iteration the auxiliary array `c` that contains the sum of the values of each column of $a[i..j][1..n]$: since the next iteration considers the subarray increased by one line, we can simply add the new added line in c.

`SP1` calls three subproblems:

SP2 a subproblem to initialise $c[1..n]$ with the line j of $a[1..m][1..n]$;

SP3 a subproblem to update the array $c[1..n]$ with the sum of the elements of line i of $a[1..m][1..n]$ and the elements of the initial c;

SP4 a subproblem to search for a subarray of sum S in the one-dimensional array, $c[1..n]$.

In this way, considering that SP2 has complexity $\mathcal{O}(n)$, SP3 has the complexity $\mathcal{O}(n)$, if we have an algorithm SP4 in $\mathcal{O}(n)$, in the worst case, the main algorithm calls 1 time SP2, $m-1$ times SP3 and m times SP4. We have the required complexity : $\mathcal{O}(m.(n+(m-1)n+nm)) = \mathcal{O}(m^2n)$.

This decomposition was followed by most groups since it should have been similar to the decomposition of the previous project, and besides, the operational point of view of the algorithm of the previous project had been introduced by the teacher.

Specifications of subproblems Students are asked to write precise specifications for each subproblem, no matter the formalism, they must just be precise and complete. Then, they have to formalise it into the language of the automatic tool. In a general way, we can observe that specifications written in the assertion language of the tool are more precise and more complete.

Here follows our decomposition into subproblems and their specifications.

SP1: Pre: $a[1..m][1..n]$, S initialised and $1 \leq j \leq m$

Post: a, S, j unchanged and

$present \Leftrightarrow (\exists\ i1 : 1 \leq i1 \leq j : (\exists k1, l1 : 1 \leq k_1 \leq l1 \leq n :$
$S = \sum_{i1 \leq ij \leq j} \sum_{k1 \leq kl \leq l1} a[ij][k_l]))$

SP2: Pre: $a[1..m][1..n]$, j initialised and $1 \leq j \leq m$

Post: a, j unchanged and

$(\forall k : 1 \leq k \leq n : c[k] = a[j][k])$

SP3: Pre: $a[1..m][1..n]$, $c[1..n]$, i initialised and $1 \leq i \leq m$

$\forall k : 1 \leq k \leq n : c[k] + a[i][k] \leq maxc$

where $maxc$ is the greatest possible value of c;

Post: a, i unchanged and

$(\forall k : 1 \leq k \leq n : c[k] = c_0[k] + a[i][k])$

SP4: Pre: $c[1..n]$, S initialised

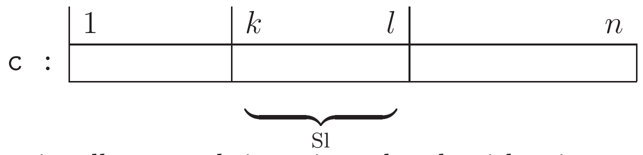
Post: c, S unchanged and

$present \Leftrightarrow (\exists k, l : 1 \leq k \leq l \leq n : S = \sum_{k \leq kl \leq l} c[kl])$

To discuss the impact of the tool on the learning of students, we detail the construction of three subproblems. First, we analyse the subproblem SP4 that searches for a subarray of sum S in a one-dimensional array; the design of this algorithm is not simple and requires to master the invariant method carefully. Then, we study the subproblem SP3 that adds the line $a[i][1..n]$, componentwize, to the array $c[1..n]$; this algorithm is simpler but needs to be carefully specified to prove that no overflow can occur. Finally, we detail the subproblem SP1 that searches in the array $a[1..j][1..n]$ for a subarray of sum S for which the last line is on line j; this problem calls different subproblems and uses the auxiliary array $c[1..n]$, which may be the source of overflow problems.

7.3.2.5 Searching for a subarray of sum S in a one-dimensional array

Here is a first glimpse:



Operationally, at each iteration, the algorithm increments the index l and updates Sl; if Sl has become greater that S, the index k is increased until $Sl \leq S$.

Invariant There are two ways to solve this problem: with two loops or one loop. Our solution consists of a decomposition into two subproblems (two loops): the main problem increments l; the subproblem increments k. So, in our implementation, after each iteration, Sl, the sum of the elements of $c[k..l]$, is the longest subarray terminating at index l such that $Sl \leq S$; besides, there are no subarray of sum S in the subarray $c[1..l-1]$. A formal version of this invariant is :

$$\begin{aligned}
&c,\ S \text{ unchanged and}\\
&1 \leq k \leq l+1 \leq n+1 \text{ and}\\
&(\forall k1, l1 : 1 \leq k1 \leq l1 < l : S \neq \sum_{k1 \leq kl \leq l1} c[kl]) \text{ and}\\
&Sl = \sum_{k \leq kl \leq l} c[kl] \text{ and}\\
&Sl \leq S \text{ and}\\
&(\forall k1 : 1 \leq k1 < k : \sum_{k1 \leq kl \leq l} c[kl]) > S)
\end{aligned}$$

Derivation of the code From this invariant, we can derive the code:

- Before any iteration, we have not yet analysed anything in the array, $[k..l]$ is an empty subarray at the beginning of the array: $[k..l] = [1..0]$ and $Sl = 0$:

  The initialisation is

  ```
  k := 1 ; l := 0 ; Sl := 0
  ```

- In the iteration, we increment l of 1 and we add $c[l]$ in Sl.

  Then, we call the subproblem in charge of reducing $c[k..l]$ if $Sl > S$;

we detail the specification of this subproblem (named `SP5`) just below; we easily see that the SP5 precondition is satisfied.

The iteration is

```
l := l+1 ; Sl := Sl + c[l] ; SP5
```

- The algorithm stops when a non empty subarray of sum S is found `(Sl == S & k < l)` or when we reach the end on the array (`l = n`).

  The halting condition is

  ```
  Sl == S & k < l | l = n
  ```

- In the closure, according to the reason why the halting condition holds, we initialise the result Boolean variable *present*:

  ```
  present := Sl = s & k < l
  ```

Specification of the subproblem SP5 The goal of this subproblem is to reduce $c[k..l]$ by increasing k such that Sl, the sum of the elements of $c[k..l]$ becomes smaller than or equal to S: l remains unchanged and k is increased but importantly, $c[k..l]$ is the greatest subarray terminating at index l such that $Sl \leq S$; in other words, k must be the smallest index such that $Sl \leq S$. A formal specification is the following:

SP5: Pre: $c[1..n]$, S initialised and

$$1 \leq k \leq l \leq n \text{ and } Sl = \sum_{k \leq k1 \leq l} c[k1]$$

Post: c, S, l unchanged and

$k_0 \leq k$ and $k \leq l+1$ and

$Sl \leq S$ and

$$Sl = \sum_{k \leq k1 \leq l} c[k1] \text{ and}$$

$$(\forall k_1 : k_0 \leq k1 < k : \sum_{k1 \leq k2 \leq l} c[k2] > S)$$

The translation in the language of the tool is provided in Figures 7.8 and 7.9.

Analysing students' solutions

Decomposition into subproblems Few groups have decomposed the subproblem SP4; but the groups who did it, did not have an ideal specification for the subproblem `SP5`.

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 0 ;
      const maxv = 1 ;
      tab   c : array[1 .. n] of minv .. m * maxv ;
      var   S : 0 .. n * m * maxv  ;

Auxiliary_variables:
      var   k : 1 .. n + 1 ;
      var   l : 0 .. n ;
      var  Sl : 0 .. m * n * maxv ;

Result_variables:
     var present : boolean ;

Precondition: initialised(1 , n : c)  & initialised(S)

Postcondition: unchanged(1 , n : c)  & unchanged(S) &&
  (present <=> (exist k : 1 <= k <= n :
               (exist l : k <= l <= n :
                  S = (sum kl : k <= kl <= l : c[kl])
                )))

Invariant: unchanged(1 , n :  c)  & unchanged(S) &
  1<= k & k <= l + 1 & l <= n &&
  (forall k1 : 1 <= k1 <= l - 1 :
     (forall l1 : k1 <= l1 <= l - 1 :
                  S != (sum kl : k1 <= kl <= l1 : c[kl]))) &
   Sl = (sum kl : k <= kl <= l : c[kl]) &
   Sl <= S &
   (forall k1 : 1 <= k1 <= k - 1 : (sum kl : k1 <= kl <= l : c[kl]) > S)


Init:  k := 1 ; l := 0 ; Sl := 0

Iter:  l := l + 1 ;
       Sl := Sl + c[l] ;
       sp(SP5.in)

Clot: present := (Sl = S) & (k <= l)

Halting_condition: ((Sl = S) & (k <= l)) | l = n

Variant: n - l
```

Figure 7.8: SP4: searching for a subarray of sum S

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv =  0;
      const maxv =  1;
      tab   c : array[1 .. n] of minv .. m * maxv ;
      var   S : 0 .. m * n * maxv ;
      var   k : 1 .. n + 1;
      var   l : 0 .. n ;
      var  Sl : 0 .. m * n * maxv ;

Precondition: initialised(1 , n : c) & initialised (S) &
      k <= l &&
      Sl = (sum k1 : k <= k1 <= l : c[k1])

Postcondition: unchanged(1, n : c) & unchanged(S) & unchanged(l) &
  k_0 <= k & k <= l + 1 &
  Sl <= S &&
  Sl = (sum k1 : k <= k1 <= l : c[k1])
  &
  (forall k1 : k_0 <= k1 <= k - 1 :
     (sum k2 : k1 <= k2 <= l : c[k2]) > S)


Invariant: unchanged(1, n : c)  & unchanged(S) & unchanged(l) &
  k_0 <= k & k <= l + 1 &&
  Sl = (sum k1 : k <= k1 <= l : c[k1]) &
  (forall k1 : k_0 <= k1 <= k - 1 :
       (sum k2 : k1 <= k2 <= l : c[k2]) > S)


Init:  skip

Iter:  Sl := Sl - c[k] ; k := k + 1

Clot:  skip

Halting_condition:  Sl <= S

Variant: (l + 1) - k
```

Figure 7.9: `SP5`: reducing `c[k..l]` by increasing `k` such that `Sl` becomes smaller than or equal to `S`

- The trend is to make *too strong preconditions*: typically, it is not necessary to say that there are no subarray of sum S in $c[1..l-1]$. It is always better to be as general as possible. This remark is subjective so that the tool is not able to give any feedback about it.
- The students made *too weak postconditions* forgetting to give some constraints on the resulting S_l. A too weak postcondition of a subproblem is systematically detected when we automatically verify the subproblem calling it.

Too weak invariants In their first version, many students have chosen a *too weak invariant*, they have forgotten important constraints on Sl: some invariants were even limited to the following constraints: $1 \leq k \leq l \leq n$ and $Sl = \sum_{k \leq k1 \leq l} c[k1]$. The tool has shown counter-examples, and students have modified their invariants. Most of the groups have not decomposed the subproblem SP4: one of their solutions is depicted in Figure 7.10. The invariant in this context is quite complex: it includes two parts of reasoning that, with our solution, we have thought in two times. Indeed, we have two subproblems and so two invariants. For some students, it is after a lot of iterations that they have found a correct invariant. Notice that the halting condition of this version is not simple.

Too complicated assertions Through the several students reports (especially for these subproblems SP4 and SP5), we can observe correct programs but with unreadable/non adapted/redundant specifications. Such complicated assertions are obtained because students modify their algorithm and their specifications iteratively until they are found correct by the tool. Here is an example of a bad assertion in a precondition for SP5:

```
Pre:
...
(forall x: 1 <= x <= k-1: (Sl + (sum y: x <= y <= k-1: c[y]) > Sl))
...
```

which is equivalent to `(k<=1 || c[k-1]>0)`. It does not seem that the user has written this non natural assertion at the first shot. In fact, this constraint exists because of a lack of generality in the specification of the subproblem: ideally, we should not mind about the subarray $c[1..k-1]$ in this subproblem.
The following assertion occurs in the postcondition of another version of SP5:

```
...
((Sl <= S &&
 (forall a : k_0 <= a <= l : ((sum s : a <= s <= l : c[s]) <= S)
                                  <=> (k <= a)))
|
((k = l) &&
 (forall x: k_0 <= x <= k : (sum s : x <= s <= l : c[s]) > S)))
```

The first part of the disjunction says that k is the smallest index in $c[k_0..l]$ such that $\sum_{k \leq k1 \leq l} c[k1] \leq S$ if it exists, the second part of the disjunction is equivalent to `(k == l && c[l] > S)`: it corresponds to the particular case where there is no $c[k..l]$ such that $\sum_{k \leq k1 \leq l} c[k1] \leq S$; in this case, $k = l$. This postcondition is not nice, we could avoid the particular case and have one general case (considering that the result array $c[k..l]$ can be empty).

We conclude that it is not always an advantage that the tool admits and understands any specification even if it is trived or cryptic, we observe a drawback of the tool: it does not enforce simplicity.

Reasoning errors The postcondition of the problem SP4 must talk about a *non empty* subarray of sum S. Many students made the mistake of finding an empty subarray when $S = 0$. Since the postcondition is generally correctly formalised:
$present \Leftrightarrow$
$(\exists k : 1 \leq k \leq n : (\exists l : \mathbf{k} \leq \mathbf{l} \leq n : S = \sum_{k \leq kl \leq l} c[kl])),$
counter-examples were provided by the tool and students have had the opportunity to change their code. Notice that the postcondition of this subproblem is generally correct; otherwise, it is detected during the verification of the calling subproblem (SP1). Importantly, to get a feedback about this error, the domain of the variable S must include the value 0.

7.3.2.6 Adding $a[i][1..n]$ to $c[1..n]$

SP3: Pre: $a[1..m][1..n]$, $c[1..n]$, i initialised and $1 \leq i \leq m$

$\forall k : 1 \leq k \leq n : c[k] + a[i][k] \leq maxc$

Post: a, i unchanged and

$(\forall k : 1 \leq k \leq n : c[k] = c_0[k] + a[i][k])$

invariant:

a, i unchanged

$1 \leq k \leq n + 1$

$c[k..n]$ unchanged

$(\forall k_1 : 1 \leq k_1 \leq k - 1 : c[k_1] = c_0[k_1] + a[i][k_1])$

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 0 ;
      const maxv = 2 ;
      tab   b : array [1 .. n] of minv .. m*maxv ;
      var   S : 0 .. m * n * maxv;

Auxiliary_variables:
      var   k : 1 .. n+1 ;
      var   l : 1 .. n+1 ;
      var   Skl : 0 .. m * n * maxv ;

Result_variables:
      var present : boolean;

Precondition:
      initialised(1, n : b) & initialised(S)

Postcondition:
      unchanged(1, n : b) & unchanged(S) &&
      (present <=> (exist k : 1 <= k <= n : (exist l : k <= l <= n :
                      S = (sum kl : k <= kl <= l : b[kl]) )) )

Invariant:
      unchanged(1, n : b) & unchanged(S) & 1 <= k & k <= l & l <= n +1 &&
      !(exist p : 1 <= p <= k-1 : (exist q : p <= q <= n :
          S = (sum pq : p <= pq <= q : b[pq]) )) &
      !(exist q : 1 <= q <= l-2 : (exist p : 1 <= p <= q :
          S = (sum pq : p <= pq <= q : b[pq]) )) &
      Skl = (sum kl : k <= kl <= l-1 : b[kl]) &
      (present <=> k != l & S = Skl)

Init: k := 1 ; l := 1 ; Skl := 0 ; present := false

Iter:
    if (Skl > S) then
        Skl := Skl - b[k] ; k := k+1
    else
        Skl := Skl + b[l] ; l := l+1
    end ;
    if (k != l && Skl = S) then present := true else present := false end

Clot: skip

Halting_condition: (present) | (l=n+1 & (k=n+1 | Skl < S))

Variant: 2*n + 2 - k - l
```

Figure 7.10: A student version of the algorithm searching for a subarray of sum S

- the initialisation is:

```
k := 1
```

- the halting condition is `k = n + 1`

- the iteration is

```
c[k] := c[k] + a[i][k] ; k := k + 1
```

  We are sure there are no out of bound of arrays since $1 \leq k \leq n$ (and $1 \leq i \leq m$) when we make an iteration and we can guarantee that $c[k]+a[i][k]$ can be assigned to $c[k]$ because the precondition expresses it explicitly.

This algorithm needs to be carefully specified to prove that no overflow can occur; moreover, most students did not pay attention to the overflow risk in the first project. Students have not thought that adding the elements of each column of the array a in an auxiliary array $c[1..n]$ could be source of error if the domain of the elements of c is not large enough to allow these additions.

The complete solution of this subproblem is given in figure 7.11.

Analysing students' solutions

Badly defined statements Most students have met the overflow problem; it has been detected by the tool. The precondition to add to prevent the overflow error has an impact on the choice of the domains of one of the global variables of the set of subproblems: the domain of c must be at least $[0..maxv * m]$: we detail this choice in the discussion of SP1, which calls iteratively SP3.

Too strong preconditions We notice that most students have written the following assertion to guarantee no overflow:

$$\forall k : 1 \leq k \leq n : c[k] \leq maxv * (m - 1)$$

Considering that each subproblem must be as general as possible, this precondition is too strong.

Inappropriate data domains Some students have not chosen judicious data domains: for this example, it is important that the constraint ($\forall k : 1 \leq k \leq n : c[k] + a[i][k] \leq maxvc$) does not reduce too much the set of possible data satisfying the precondition. Indeed, if the set of possible data is too small, it cannot be representative to have confidence in the tool feedback, because it may miss the discovery of counter-examples, testifying a false correctness of the algorithm.

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 45 ;
      const maxv = 47 ;
      tab   a : array [1 .. m][ 1.. n] of minv .. maxv ;
      var   i : 0 .. m ;
      tab   c : array[1 .. n] of minv .. m * maxv ;

Auxiliary_variables:
      var   k : 0 .. maxint ;

Result_variables:

Precondition: initialised(a) & 1 <= i & i <= m &&
  (forall k : 1 <= k <= n : c[k] + a[i][k] <=  m * maxv)

Postcondition: unchanged(a)  & unchanged(i) &&
  (forall k : 1 <= k <= n : c[k] = c_0[k] + a[i][k])

Invariant: unchanged(a)  & unchanged(i) 1 <= k & k <=n + 1 &&
  (forall k1 : 1 <= k1 <= k-1 : c[k1] = c_0[k1] + a[i][k1])
  & unchanged (k , n : c)

Init:  k := 1

Iter:  c[k] := c[k] + a[i][k] ;
       k : = k + 1

Clot: skip

Halting_condition:   k = n + 1

Variant: n + 1 - k
```

Figure 7.11: SP3: adding $a[i][1..n]$ to $c[1..n]$

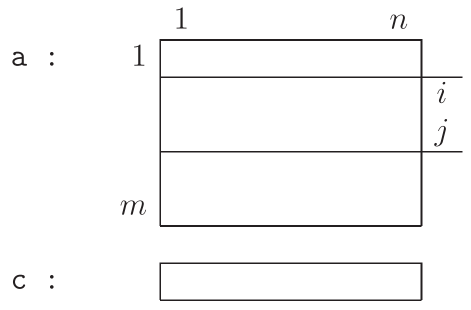
Too weak invariants Some students have forgotten to say that $c[k..n]$ is unchanged; since the tool has given counter-examples, they have improved their invariant. Notice that very few students use the hypothesis that $c[k..n]$ is unchanged to prove the correctness of $\{Inv$ and not $H\}$ $Iter$ $\{Inv\}$, using symbolic execution. Thanks to the tool, they manage to write a correct invariant but it did not help them to actually understand the necessity of the assertion required by the tool.

Redundant assertions Some students have expressed in their invariant that the array $c[k..n]$ is unchanged and have also recalled the precondition on this array c.

7.3.2.7 Searching for a subarray of sum `S` in `a[1..j][1..n]` such that last line is on line `j`

To decompose the problem into subproblems, we already had an idea of the general situation which is true before and after every iteration. For this subproblem, we need an auxiliary array, say $c[1..n]$, that allows us to reduce the problem of finding a subarray of sum S in a two-dimensional array to a one-dimensional problem. At each iteration, this subarray contains one more line.

The `SP1` invariant is:



a is unchanged
j is unchanged
$1 \leq i \leq j$
present is true if a subarray of sum S exists in the subarray $a[i..j][1..m]$; *present* is false otherwise.
$(\forall\ k : 1 \leq k \leq n : c[k] = \sum_{i \leq l \leq j} a[l][k])$

- If we initialise the auxiliary variable i with j, corresponding to the first line to be analysed; if we initialise the array $c[1..n]$ with the values of line i; if we make a first search in $c[1..n]$ for a subarray of sum S (calling `SP4` which update the Boolean variable *present* according to its SP4 postcondition), the invariant holds.

So, the initialisation is

```
i := j ; SP2 ; SP4 ;
```

- We stop the search when we have analysed the whole array $a[1..j][1..n]$ (i.e., $i = 1$) or when a subarray of sum S is found (i.e., $present = true$).

  The halting condition is

  ```
  present | i = 1
  ```

- We first increase the height of the subarray that we search by decrementing i,

  we get $1 \leq i \leq m$.

  We add the line i in $c[1..n]$ (we focus on the SP3 precondition satisfaction in the pedagogical observations).

  The resulting array c contains the sum (column after column) of the lines of $a[i..j][1..n]$.

  We search for a subarray of sum S in c;

  present is updated accordingly.

  The iteration is

  ```
  i := i - 1 ; SP3 ; SP4
  ```

The version in the tool format is in Figure 7.12.

Analysing students' solutions

The choice of the domain of $c[1..n]$ Intuitively, thinking that we have to add maximum m lines of an array $a[1..m][1..n]$ for which the maximum value of the elements is $maxv$, the greatest value of the elements of c is at least $maxvc = m * maxv$. Notice that this problem was not considered by most students when they have made the Java project that also needed an auxiliary array containing a maximum of m lines of integers. Thanks to the tool feedback, the students have improved their solution.

More formally, let us focus on a part of `SP3` precondition:

$$\forall k : 1 \leq k \leq n : c[k] + a[i][k] \leq maxvc$$

The `SP1` invariant says that

$$\begin{array}{l} 1 \leq j \leq m \text{ and} \\ 1 < i \leq m \text{ and} \\ (\forall\ k : 1 \leq k \leq n : c[k] = \sum_{i \leq l \leq j} a[l][k]) \end{array}$$

Considering that this invariant holds, it implies that

$$(\forall\ k : 1 \leq k \leq n : c[k] \leq \sum_{2 \leq l \leq m} maxv)$$

i.e., $(\forall\ k : 1 \leq k \leq n : c[k] \leq (m-1) * maxv)$
which implies that $(\forall\ k : 1 \leq k \leq n : c[k] + a[i][k] \leq (m-1) * maxv + maxv)$

To guarantee that precondition $\forall k : 1 \leq k \leq n : c[k] + a[i][k] \leq maxvc$ is true when we call `SP3`, $maxvc$ must be at least equal to $m * maxv$.

Calls of subproblems Because this problem calls subproblems, non complete postconditions of the subproblems called have been discovered during the verification of this problem. The most frequent error was to omit to say that a variable was unchanged.

We do not construct the simple subproblem SP2 that initialises the array $c[1..n]$ with a line i of the array $a[1..m][1..n]$; a complete version is in Figure 7.13. The main subproblem has been analysed during the decomposition into subproblems; we provide a complete version in Figure 7.14.

7.3.2.8 Putting everything together

After having distributed the different subproblems and shared the global variables and the formal specifications of subproblems, each student has built his algorithm individually. Notice that the tool allows that a subproblem is only specified (declaration-precondition-postcondition); in this way the students are able to use the specification of a called subproblem even if it is not solved yet.

For each subproblem, individually, the student has first chosen by himself appropriate variable domains. When the group has put everything together, the students had to fix the variable domains in a global way. Theoretically, the sharing should not require effort if the preparation of the work, i.e., the decomposition into subproblems and the formalisation of the specifications was correctly done.

7.3.2.9 Evaluation of the project

Good reports In general, we are happy with the interest of the students in this project. We have received forty group reports; only one group has given up the project. The work was globally done conscientiously; most students have reached a solution, even if some students have spent a lot of time to reach it.

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 45 ;
      const maxv = 47 ;
      tab   a : array [1 .. m][ 1.. n] of minv .. maxv ;
      var   S : 0 .. maxint ;
      var   j : 0 .. m ;

Auxiliary_variables:
      tab   c : array[1 .. n] of minv .. m * maxv ;
      var   i : 0 .. m + 1 ;

Result_variables:
      var  present : boolean ;

Precondition: initialised(a) & 1 <= j

Postcondition: unchanged(a)  & unchanged(j) &&

 (present <=> (exist i1 : 1 <= i1 <= j :
             (exist k1 : 1 <= k1 <= n : (exist l1 : k1 <= l1 <= n :
                 S = (sum ij : i1 <= ij <= j :
                     (sum kl : k1 <= kl <= l1 : a[ij][kl]))
               )))


Invariant: unchanged(a)  & unchanged(j) && 1 <= i & i <= j  &&

 (forall k : 1 <= k <= n : c[k] = (sum ij : i <= ij <= j : a[ij][k])) &

 (present <=> (exist i1 : i <= i1 <= j :
             (exist k1 : 1 <= k1 <= n : (exist l1 : k1 <= l1 <= n :
                 S = (sum ij : i1 <= ij <= j :
                     (sum kl : k1 <= kl <= l1 : a[ij][kl]))
               )))


Init:  i := j ;
       sp(SP2.in) ;
       sp(SP4.in);

Iter:  i := i - 1 ;
       sp(SP3.in)
       sp(SP4.in) ;

Clot: skip

Halting_condition:   present | i = 1

Variant: i
```

Figure 7.12: SP1: searching for a subarray of sum S in $a[1..j][1..n]$ such that the last line is on line j.

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 0 ;
      const maxv = 1 ;
      tab   a : array [1 .. m * n] of minv .. maxv ;
      var   i : 0 .. m ;

Auxiliary_variables:
      var   k : 0 .. n ;

Result_variables:
      tab   c : array[1 .. n] of 0 .. m * maxv ;

Precondition: initialised(1, m*n : a) & 1 <= i

Postcondition: unchanged(1, m*n : a)  & unchanged(i) &&
  (forall k : 1 <= k <= n : c[k] = a[n * (i-1) + k])

Invariant: unchanged(1 , m*n : a)  & unchanged(i) &&
  (forall k1 : 1 <= k1 <= k : c[k1] = a[n * (i-1) + k1])


Init:  k := 0

Iter:  k := k + 1 ;
        c[k] := a[n * (i-1) + k]

Clot: skip

Halting_condition:   k = n

Variant: n - k
```

Figure 7.13: SP2: initialising the array $c[1..n]$ with $a[i][1..n]$;

```
Data: const n : 1 .. 3 ;
      const m : 1 .. 3 ;
      const minv = 45 ;
      const maxv = 47 ;
      tab   a : array [1 .. m][ 1.. n] of minv .. maxv ;
      var   S : 0 .. maxint ;

Auxiliary_variables:
      var   j : 0 .. m ;

Result_variables:
      var  present : boolean ;

Precondition: initialised(a)

Postcondition: unchanged(a)  &&

  (present <=> (exist i : 1 <= i <= m : (exist j : i <= j <= m :
               (exist k : 1 <= k <= n : (exist l : k <= l <= n :
                  S = (sum ij : i <= ij <= j :
                       (sum kl : k <= kl <= l : a[ij][kl]))
                )))

Invariant: unchanged(a)  &&

  (present <=> (exist i1 : 1 <= i1 <= j : (exist j1 : i1 <= j1 <= j :
               (exist k1 : 1 <= k1 <= n : (exist l1 : k1 <= l1 <= n :
                  S = (sum ij : i1 <= ij <= j1 :
                       (sum kl : k1 <= kl <= l1 : a[ij][kl]))
                )))

Init:  j := 0 ; present := false

Iter:  j := j + 1 ;
       sp(SP1) ;

Clot: skip

Halting_condition:   present | j = m

Variant: m - j
```

Figure 7.14: Main problem: searching for a non empty subarray of sum S in $a[1..m][1..n]$

Students performance We observe two groups of students who did not have any difficulty to solve their algorithms using an invariant: no error was found by the tool The reports and the discussions we had with students show that the tool has been efficient to warn them that their reasoning was false. Their algorithm was not correct at first time, but, thanks to the tool they had the opportunity to modify their algorithm. Finally, two groups have never reached a correct solution. It is important to notice that this categorization of the students is deeply bound to their orientation: the students in engineering (especially those in applied mathematics) have a better abstraction level and a higher rigour than the bachelor students as well as the graduates in computer science.

Manual proof In the reports, we observe that the errors are nearly never discovered through the manual proof, but rather by using the tool. Students do not make their manual proofs correctly, they keep in mind the operational reasoning instead of thinking in a declarative way.

Efficiency of the tool Many solutions (especially for subproblems `SP4` and `SP3`) have not been correct in one shot: for `SP4`, few students are able to write a complete invariant at first shot; for `SP3`, many students have met problems of out of domains errors. But in most cases, the students have reached a correct solution: we think that the feedback of the tool has allowed better reports; manual proofs have also been improved. Many students have overcome their difficulties.

Motivation of the students We think that for many groups, the motivation was great. Nonetheless, motivation is not synonymous with the interest to master the structured programming method. Some students have spent a lot of time to reach a correct solution: they use the tool with an approach similar to the process of “fixing” bugs in programs by trial and error.

7.4 Conclusions

The tool helps to understand the method Clearly, the tool allows the students to understand some notions that seem difficult. We have observed that, no matter the number of exercises on paper we do with them, the students (excepted from the more “mathematician” students) begin to understand the notions when they use the tool: for example, the fact that, when we consider $\{Inv \text{ and not } B\}$ *iter* $\{Inv\}$, we cannot refer to the initialisation, has been a revelation for many students at the first lab session using the tool. The concrete counter-examples were the source of their understanding. However, the experimentation shows that students needs explanation from the teacher to understand the first time the counter-examples.

The tool enforces the use of the structured programming method Indeed, the user cannot avoid the decomposition into subproblems and the use of invariants for building loops. There are two main reasons for that: the format of the algorithm and the automatic verification that strictly follows the principles of the structured programming method. Each specification and each invariant must be extremely precise and in agreement with the code.

The tool helps to understand the role of a complete specification Since the automatic verification is based on specifications, and does not use the code of called subproblems, the specifications must be definitely complete. Counter-examples can help the user to understand why his specification is not strong enough.

The tool helps to learn to write formal specifications A substantial amount of students have difficulties to express their meaning in a pictorial way and still more in a pure formal way. Encoding the specification formally in the tool is a good exercise for them; besides, counter-examples can help the user to understand that he did not formally describe his thoughts.

The tool contributes to improve the level of abstraction The programming method requires a good level of abstraction, discouraging the students during the learning phase. Since the tool enforces the user to use the structured programming method, since the students are motivated, since the tool gives concrete counter-examples when the reasoning is not correct, by experimenting several times the tool, the level can be better. Obviously, the tool cannot solve the general problem of lack of abstraction of our computer science students.

The tool captivates the attention When we make exercises together, using the board, a large part of students listen passively; we have made the experiment with the construction of the algorithm computing the number of common values in two strictly increasing arrays. This exercise has been solved on the board. The day we made the same exercise using the tool, many students hadforgotten the board session involving it. Using the tool, students work more intensively.

The tool can be used by trial and error Both experiments have shown that the tool does not prevent the students from solving their problem by trial and error. Even when they approximately understand the method, the trend is to iterate modifications in the invariant and in the code without thinking enough. They should stop this process, take a sheet of paper and reason on picture before encoding the modification. It seems too tempting to test the correctness using the tool. We have serious doubts on the actual chronology that students have followed to solve their problem: most of them have probably proved the correctness of their algorithm using the tool before proving it manually.

The tool cannot enforce simplicity We have also observed that students do not necessarily think about simple things: simple specifications, simple decompositions into subproblems, simple invariants, etc.

Our tool is useful and efficient, but it is only a tool. The construction of program is a complex work, and requires a lot of personal involvement. The tool is a support to learning, but the learning has to be achieved by the student, anyhow.

Chapter 8

Conclusion

During these long years of research, we have focused on tools that can help us teach the structured programming method.

The first approach was to see if some existing tools were adapted to our needs. Indeed, a few verification tools are available on the market, ready to be used. After approaching the most promising ones, we had to admit the difficulty of getting an existing software that is appropriate enough for our specific pedagogical objectives since their objectives are not ours. Using verification tools like ESC/Java2 or SPARK, the student cannot get a precise feedback: too often ESC/Java2 gives false warnings, and sometimes it forgets errors; too often SPARK needs the use of its interactive proof checker which requires too many efforts and too much expertise from the user to be able to manipulate it. Since softwares based on theorem proving either require extensive human guidance, or are limited to verify simple properties, we have looked at techniques of exhaustive verification, model-checking. We have experimented the use of a model-checker, SMV. It has the advantage to be automatic and to give a precise feedback to the user but it cannot be used as it is, because the work required to formalise these properties is well beyond what can be asked to students learning the methodology. Software model-checkers have not been deeply investigated, but, to our knowledge, they are limited to handle specific properties of programs but not fully complete specifications.

Since we have not found the perfect tool for our needs, our solution was to develop our own tool, completely adapted to our objectives. It appears essential that an appropriate tool should give a precise feedback about the correctness of the algorithms. It seems also important that the tool admits an expressive assertion language to allow the user to write its specifications straightforwardly. Besides, the tool must enforce the use of the structured programming method.

Having made precise the requirements for such a tool, we quickly found out that constraint programming techniques are convenient to implement it. We have chosen the multiparadigm language Oz because it is a powerful language that includes all programming mechanisms that are needed to reach our goals. Besides, working at UCL in the same department as the Belgian Mozart/Oz pool, we have been easily convinced to use it.

Thus, we have implemented in Oz/Mozart a tool that corresponds to our requirements. This tool gives optimistic results in the sense that it is fully automatic and it is able to give precise feedback to the student in moderate times. To evaluate our tool, it was useful to analyse the impact of such a tool on the behaviour, the motivation and the understanding of the students who learn this programming method. An interesting application of our research is in the actual use of our tool in a real-life teaching world, with real students: our tool is used as a support to the "Program Conception Methods" course addressed to the computer science students in the third year of their studies. Experimentation has shown that the tool is completely useful in a regular academic context, that it really motivates students and helps them to learn the taught methodology, and that its use was reinforced. However, our tool stays only a tool, it is a way to learn. The role of a teacher remains important: students may need explanations, and besides the tool does not evaluate subjective points like simplicity or the adequacy of the decomposition into subproblems. Another drawback which is inherent in any tool is that it can be used by trial and error.

Of course, some improvements could be useful. The assertion language is expressive, but after experimenting the tool with a lot of exercises, we have faced several assertion examples for which it would have been handy to provide a richer syntax. Such an enhancement is not difficult to implement. In this same idea, it would have been nice to allow the user to define himself some predicates, used as macros. A functionality that would not be too difficult to add is giving the possibility to the student to check the correctness of $\{P\}\ S\ \{Q\}$ for any step of the algorithm. It would also be interesting to be able to check the correctness of a implication or equivalence between two assertions: it could allow the user to check a proof for which he manually transforms assertions using sp or wp method; it could allow the user to check the correctness of his specification checking the equivalence with the teacher's specification. For problems decomposed into subproblems, it would be interesting to parameterize the variable domains. A non trivial work would be to introduce the support of parameters in the procedure definitions. In a general way, saving the history of the elaboration of the algorithms would be a nice feature, so that the student can check the different steps he had to go through to complete his task. In some scenarii, we could simply try to prevent the student to overuse the trial and error process, by limiting the maximum number of verifications.

Globally, the tool was well accepted by our students. It was a great help to teach the structured programming method. The tool is a support to learning, but the learning has to be achieved by the student, anyhow. Still, we are realistic enough to think that our students will not systematically think through that method. If we have contributed to a better understanding of the method, it is already a big step forward.

Appendix A

Existing Tools

A.1 ESC/Java2

A.1.1 The Binary Search

The following algorithm provides a variant, to let ESC/Java2 proving termination of the algorithm. Notice that without the variant, ESC/Java2 is not able to detect if the loop does not terminate.

```
/*@requires a != null  &&
           (\forall int i,j ; 0 <= i & i < j & j <= a.length ;
                                a[i] <= a[j]);
  @ensures \result <==> (\exists int j ; 0 <= j & j < a.length ;
                                          a[j] == x) ;
  @*/

  public /*@ pure @*/ static boolean dicho(int[] a, int x)
  {
      int n = a.length;
      int m;
      int g = 0;
      int d = n;
      boolean b = false;
      int y = 0;

/*@loop_invariant 0 <= g & g <= d & d <= a.length ;
  @loop_invariant (\forall int i ; 0 <= i & i < g ;
                                  a[i] < x) ;
  @loop_invariant (\forall int i ; d <= i & i < a.length ;
                                  a[i] > x) ;
  @loop_invariant b ==> (\exists int j ; 0 <= j & j < a.length;
                                         a[j] == x) ;
  @loop_invariant b <==> y==1 && !b <==> y == 0 ;
  @*/

  //@decreasing d - g - y ;
```

```
    while ((b == false) & (g < d)){

        m = (g + d) / 2 ;
        if (a[m] < x) g = m + 1 ;
        else if (a[m] > x) d = m ;
        else {b = true ; y = 1 ;}
    }

    return b;
}
```

A.1.2 The Next Permutation

Subproblem 2 The first subproblem is completely detailed in Section 3.1; let us display the second subproblem, it is formally specified to be verified by ESC/Java2.

```
/*@requires a != null && 0 <= i && i <= a.length - 2;
  @requires (\forall int k ; 0 <= k && k < a.length - 2 - i ;
                              a[k + 1 + i] >= a[ k + 2 + i]) ;
  @requires a[i] < a[i + 1] ;
  @ensures i + 1 <= \result & \result <= a.length - 1 ;
  @ensures a[\result] > a[i];
  @ensures \result == a.length - 1 || a[\result + 1] <= a[i] ;
  @*/

  public /*@ pure @*/ static int sp2(int[] a, int i) {

        int j = i+1;

/*@loop_invariant  i + 1 <= j && j <= a.length - 1;
  @loop_invariant  (\forall int k ; 0 <= k && k <= j - i - 1 ;
                                     a[k + i + 1] > a[i]) ;
  @*/

        while ((j != a.length - 1) && (a[j + 1] > a[i]) ) j++;

        return j;
}
```

The implementation of this subproblem is quite easy, when the pre and post-conditions are well thought. In this exercise, all free variables of quantified assertions are ranged for 0 and ESC/Java2 is able to verify this algorithm. One can notice that ESC/Java2 does not mind if an assertion is not well defined, for example, with

```
 @ensures \result == a.length - 1 | a[\result + 1] <= a[i] ;
```

instead of the last line in postcondition.

Subproblem 3 Here follows the third subproblem; this algorithm manipulates many index values, and the requirements of Simply do not make the formalisation easier .

```
/*@requires a != null && a.length > 0 & 0 <= i & i < a.length-1;
  @ensures (\forall int k ; 0 <= k & k <= i ;
                             \old(a[k]) == a[k]);
  @ensures (\forall int j; 0 <= j && j < (a.length - 1 - i) / 2 ;
             a[j + i + 1] == \old(a[a.length - 1- j ])
          && \old(a[j + i + 1]) == a[a.length - 1 - j]) ;
  @*/

    public static void sp3(int[] a, int i) {

        int l = i;
        int temp;

/*@loop_invariant i <= l  && l <= i + ((a.length - 1 - i) / 2) ;
  @loop_invariant (\forall int k ; 0 <= k & k <= i ;
                                    \old(a[k]) == a[k]);
  @loop_invariant (\forall int j ; 0  <= j & j < l- i -1 ;
                      a[j + i + 1] == \old(a[(a.length - 1 - j)])
                   && \old(a[j + i + 1]) == a[(a.length - 1 - j)]);
  @loop_invariant (\forall int k; l + 1  <= k && k <= a.length - 1 + i - l ;
                                   a[k] == \old(a[k])) ;
  @*/

        while (l < i + ((a.length - 1 - i) / 2)) {
            l++ ;
            temp = a[l] ;
            a[l] = a[a.length - 1 - (l - i -1)] ;
            a[a.length - 1 - (l - i - 1)] = temp ;
        }

    }
```

This algorithm seems to be checked by ESC/Java2; but the postcondition is not strong enough; ESC/Java2 is not able to prove the following postcondition which is complete

```
@ensures (\forall int k ; 0 <= k & k <= (a.length - 1 - i) / 2
;
             a[k + i + 1] == \old(a[a.length - 1- k])
          && \old(a[k + i + 1]) == a[a.length - 1 - k]) ;
```

This assertion considers the middle element of the subarray $a[i{+}1..a..\text{length}-1]$ when the number of elements of this subarray is odd. A nicer way to formalise it is

```
@ensures (\forall int k ; 0 <= k & k <= a.length - 2 - i   ;
           a[k + i + 1] == \old(a[a.length - 1- k ]));
```

but it does not seem to be adapted to ESC/Java2.

Main algorithm To generate the verification conditions, ESC/Java2 uses the specifications of the subproblems. One can try to help Simplify by inserting intermediate assertions but, anyway, this exercise is really too difficult. The pre and post conditions are formalised in Section 3.1.3.

```
public static boolean permSuiv(int[] a, int[] c) {

        int i = sp1(a);

        if (i == -1) return false;
        else {

/*@assert 0 <= i && i <= a.length -2;
  @assert  (\forall int k ; 0 <= k && k < a.length - 2 - i;
                               a[k + 1 + i] >= a[k + 2 + i]);
  @assert a[i] < a[i+1];
  @*/

           int j = sp2(a, i);

/*@assert (i + 1 <= j & j <= a.length - 1);
  @assert a[j] > a[i];
  @assert j == a.length-1 || a[j + 1]<= a[i];
  @*/

           int temp = a[i];
           a[i] = a[j];
           a[j] = temp;

/*@assert a[i] == \old(a[j]);
  @assert a[j] == \old(a[i]);
  @assert (\forall int k; 0 <= k & k <= a.length-1 ;
                            (k != i && k != j) ==> \old(a[k]) == a[k]);
  @*/

           sp3(a, i);

/*@assert (\forall int k; 0 <= k & k < i ; \old(a[k])==a[k]);
  @assert a[i]==\old(a[j]);
  @assert (\forall int k; 0 <= j && j < (a.length - 1 - i) / 2 ;
             (k + i + 1 != j && a.length - 1- k != j)  ==>
             (a[k + i + 1] == \old(a[a.length - 1- k])
          && \old(a[k + i + 1]) == a[a.length - 1 - k])) ;
  @*/
           return true;
        }
}
```

As consequence, when the user wants to overcome the tool limits, he may spent a lot of time to try to prove the correctness of algorithms without success.

A.2 SPARK

A.2.1 Proofs of User-defined Rules for the Exponentiation

We prove `X**Y * X**Z = X**(Y+Z)` by induction, using the definition given by

```
exp(4):  X ** Y        &   X * (X ** (Y - 1)) are interchangeable if [ Y > 0].
exp(5):  X ** (Y + 1)  &   X * (X ** Y)       are interchangeable if [ Y >= 0].
```

The following clause is written in a `.vcg` file.

```
H1:    z >= 0 .
H2:    y >= 0 .
       ->
C1:    x ** y * x ** z = x ** (y + z).
```

And types of `x,y,z` are specified in a `.fld` file.
We display the `.plg` file containing the Interactive Proof log:

```
STEP 1
Command: newvc

H1:  z >= 0
H2:  y >= 0
-->
  C1:  x ** y * x ** z = x ** (y + z)


STEP 2
Command: prove
Attempt to prove x ** y * x ** z = x ** (y + z)
by induction on z (base case: z = 0)
Commence PROOF BY INDUCTION attempt
Entering new proof frame (DEPTH 1) ...
  All current conclusions withdrawn
  >>> New goal C1: int_z_1 >= 0 and
                   for_all(int_z_2 : integer, 0 <= int_z_2 and int_z_2 <= int_z_1
                      -> x ** y * x ** int_z_2 = x ** (y + int_z_2))
                        -> x ** y * x ** (int_z_1 + 1) = x ** (y + (int_z_1 + 1))

  STEP 3
  Command: prove
  Commence PROOF BY IMPLICATION attempt
  Entering new proof frame (DEPTH 2) ...
    All current conclusions withdrawn
    *** New H3: int_z_1 >= 0
    *** New H4: for_all(int_z_2 : integer, 0 <= int_z_2 and int_z_2 <= int_z_1
                 -> x ** y * x ** int_z_2 = x ** (y + int_z_2))
    >>> New goal C1: x ** y * x ** (int_z_1 + 1) = x ** (y + (int_z_1 + 1))
```

```
STEP 4
Command: unwrap(h # 4)
*** New H5: 0 <= int_INT_Z_2_1 and int_INT_Z_2_1 <= int_z_1
            -> x ** y * x ** int_INT_Z_2_1 = x ** (y + int_INT_Z_2_1)

STEP 5
Command: instantiate
*** New H5: 0 <= int_z_1 and int_z_1 <= int_z_1
            -> x ** y * x ** int_z_1 = x ** (y + int_z_1)

STEP 6
Command: infer
Successful inference with rule: inference(2)
  Proved subgoal: 0 <= int_z_1 and int_z_1 <= int_z_1
                  -> x ** y * x ** int_z_1 = x ** (y + int_z_1)
  Proved subgoal: 0 <= int_z_1 and int_z_1 <= int_z_1
Therefore x ** y * x ** int_z_1 = x ** (y + int_z_1)
*** New H6: x ** y * x ** int_z_1 = x ** (y + int_z_1)

STEP 7
Command: replace(c # 1)
Successful substitution with rule: exp(5)
  Proved subgoal: int_z_1 >= 0
Allowing substitution of x * x ** int_z_1
for x ** (int_z_1 + 1)
>>> New goal C1: x ** y * (x * x ** int_z_1) = x ** (y + (int_z_1 + 1))
Successful substitution with rule: assoc(2)
  (unconstrained rule: no subgoals)
Allowing substitution of y + int_z_1 + 1
for y + (int_z_1 + 1)
>>> New goal C1: x ** y * (x * x ** int_z_1) = x ** (y + int_z_1 + 1)

STEP 8
Command: infer
Successful inference with rule: inequals(9)
  Proved subgoal: y >= 0
  Proved subgoal: int_z_1 >= 0
Therefore y + int_z_1 >= 0
*** New H7: y + int_z_1 >= 0

STEP 9
Command: replace(c # 1)
Successful substitution with rule: exp(5)
  Proved subgoal: y + int_z_1 >= 0
Allowing substitution of x * x ** (y + int_z_1)
for x ** (y + int_z_1 + 1)
>>> New goal C1: x ** y * (x * x ** int_z_1) = x * x ** (y + int_z_1)
```

```
    STEP 10
    Command: replace(c # 1)
    Successful substitution with rule: assoc(4)
      (unconstrained rule: no subgoals)
    Allowing substitution of x ** y * x * x ** int_z_1
    for x ** y * (x * x ** int_z_1)
    >>> New goal C1: x ** y * x * x ** int_z_1 = x * x ** (y + int_z_1)
    Successful substitution with rule: commut(2)
      (unconstrained rule: no subgoals)
    Allowing substitution of x * x ** y
    for x ** y * x
    >>> New goal C1: x * x ** y * x ** int_z_1 = x * x ** (y + int_z_1)

    STEP 11
    Command: replace(c # 1)
    Successful substitution with rule: assoc(3)
      (unconstrained rule: no subgoals)
    Allowing substitution of x * (x ** y * x ** int_z_1)
    for x * x ** y * x ** int_z_1
    >>> New goal C1: x * (x ** y * x ** int_z_1) = x * x ** (y + int_z_1)

    STEP 12
    Command: replace(c # 1)
    Successful substitution with rule: eq(1)
      Proved subgoal: x ** y * x ** int_z_1 = x ** (y + int_z_1)
      Met constraint: x ** y * x ** int_z_1 \= x ** (y + int_z_1)
    Allowing substitution of x ** (y + int_z_1)
    for x ** y * x ** int_z_1
    >>> New goal C1: x * x ** (y + int_z_1) = x * x ** (y + int_z_1)

    STEP 13
    Command: done
    *** Proved C1: x * x ** (y + int_z_1) = x * x ** (y + int_z_1)
    Exiting current proof frame (DEPTH 2)
  *** New H3: int_z_1 >= 0
         and for_all(int_z_2 : integer, 0 <= int_z_2 and int_z_2 <= int_z_1
                 -> x ** y * x ** int_z_2 = x ** (y + int_z_2))
                     -> x ** y * x ** (int_z_1 + 1) = x ** (y + (int_z_1 + 1))
  *** Proved C1: int_z_1 >= 0
            and for_all(int_z_2 : integer, 0 <= int_z_2 and int_z_2 <= int_z_1
                   -> x ** y * x ** int_z_2 = x ** (y + int_z_2))
                      -> x ** y * x ** (int_z_1 + 1) = x ** (y + (int_z_1 + 1))
  Exiting current proof frame (DEPTH 1)
*** New H3: x ** y * x ** z = x ** (y + z)
*** Proved C1: x ** y * x ** z = x ** (y + z)
*** Proved all conclusions

*** PROVED VC procedure_algo_2
```

```
The following rules were used in proving the above VC:
          c:/praxis/bin/../lib/checker/rules/ARITH.RUL::assoc(2)
          c:/praxis/bin/../lib/checker/rules/ARITH.RUL::assoc(3)
          c:/praxis/bin/../lib/checker/rules/ARITH.RUL::assoc(4)
          c:/praxis/bin/../lib/checker/rules/ARITH.RUL::commut(2)
          c:/praxis/bin/../lib/checker/rules/FDLFUNCS.RUL::exp(5)
          c:/praxis/bin/../lib/checker/rules/NUMINEQS.RUL::inequals(9)
          c:/praxis/bin/../lib/checker/rules/SPECIAL.RUL::eq(1)
          c:/praxis/bin/../lib/checker/rules/SPECIAL.RUL::inference(2)
```

A.3 SMV

A.3.1 Complete SMV Script for the Insertion sort

```
MODULE main
VAR
  a : array 1..4 of  1..4;
  a0 : array 1..4 of 1..4;  --to memorise values at the precondition state
  i: 0..5;
  state : {state-iter1,state-iter2,state-preSP,state-postSP};
  perm_a_a0 : perm(a,a0,i);          --permut(a,a0,1,i,1,i)
  perm_a_a0N : permN(a,i+1);         --permut(a, next(a), 1,i+1,1,i+1)
  increasing_a: increasing(a,i);     --forall j:1<=j<i: a[j]<=a[j+1]
  increasingN_a: increasingN(a,i+1); --forall j:1<=j<=i: next(a[j])<= next(a[j+1])
  unchanged_a: unchanged(a,a0,i);    --unchanged(i+1,4:a)
  unchangedN_a: unchangedN(a,i+1);   --unchanged(i+2,4:next(a))
ASSIGN
--relations between program points
  next(state):= case state = state-iter1 & i<4 : state-preSP;
                     state = state-preSP : state-postSP;
                     state = state-postSP : state-iter2;
                     1: state;
                esac;
--i:=i+1 in Iter
  next(i):= case state=state-postSP &i<5: i+1;
                 1:i;
            esac;

--a[i] is unchanged unless the SP is called
  next(a[1]):= case state = state-preSP : 1..4;
                     1: a[1];
               esac;
  next(a[2]):= case state = state-preSP : 1..4;
                     1: a[2];
               esac;
  next(a[3]):= case state = state-preSP : 1..4;
                     1: a[3];
               esac;
  next(a[4]):= case state = state-preSP : 1..4;
                     1: a[4];
               esac;
--a0 stays unchanged
  next(a0[1]):= a0[1];
  next(a0[2]):= a0[2];
  next(a0[3]):= a0[3];
  next(a0[4]):= a0[4];

--SP call
  TRANS
     (state= state-preSP & assert-preSP) -> (assert-postSP)
```

```
DEFINE assert-preSP :=  0<=i & i< 4 & increasing_a.result;
--0<=i<4 && forall j:1<=j<i: a[j]<=a[j+1]

DEFINE assert-postSP:= (unchangedN_a.result  & increasingN_a.result
                             & perm_a_a0N.result
                       );

DEFINE assert-iter1:= (i<4 & increasing_a.result  & unchanged_a.result
                       & perm_a_a0.result
                     ) ;
DEFINE assert-iter2:= (i<=4 & increasing_a.result  & unchanged_a.result
                       & perm_a_a0.result
                     ) ;

SPEC ((state=state-iter1) & assert-iter1) ->
       AG (state = state-iter2 -> assert-iter2)
SPEC ((state=state-iter1) & assert-iter1) -> AF(state = state-iter2)

MODULE increasing(a,i)
--0<=i<= 4 && forall j: 1<=j<=i-1: a[j]) <= a[j+1]
DEFINE result:= (i=0) |
                (i=1) |
                (i=2 & a[1]<=a[2]) |
                (i=3 & a[1]<=a[2] & a[2]<=a[3]) |
                (i=4 & a[1]<=a[2] & a[2]<=a[3] & a[3]<=a[4]);

MODULE increasingN(a,i)
--0<=i<= 4 && forall j: 1<=j<=i-1: next(a[j]) <= next(a[j+1])
DEFINE result:= (i=0) |
                (i=1) |
                (i=2 & next(a[1])<=next(a[2])) |
                (i=3 & next(a[1])<=next(a[2]) & next(a[2])<=next(a[3])) |
                (i=4 & next(a[1])<=next(a[2]) & next(a[2])<=next(a[3]) &
                       next(a[3])<=next(a[4]));

MODULE unchanged(a,a0,i)
-- 0<=i<= 4 && forall j: i+1<=j<=4: a[j] =a0[j]
DEFINE result:= (i=4) |
                (i=3 & a[4]=a0[4]) |
                (i=2 & a[4]=a0[4] & a[3]=a0[3]) |
                (i=1 & a[4]=a0[4] & a[3]=a0[3] & a[2]=a0[2]) |
                (i=0 & a[4]=a0[4] & a[3]=a0[3] & a[2]=a0[2] & a[1]=a0[1]);

MODULE unchangedN(a,i)
-- 0<=i<= 4 && forall j: i+1<=j<=4: next(a[j]) =a[j]
DEFINE result:= (i=4) |
                (i=3 & next(a[4])=a[4]) |
                (i=2 & next(a[4])=a[4] & next(a[3])=a[3]) |
                (i=1 & next(a[4])=a[4] & next(a[3])=a[3] & next(a[2])=a[2]) |
                (i=0 & next(a[4])=a[4] & next(a[3])=a[3] & next(a[2])=a[2] &
                       next(a[1])=a[1]);

MODULE count(a,x,i)
-- #x dans a[1..i]
DEFINE result := (a[1]=x & 1<=i) + (a[2]=x & 2<=i) +
                 (a[3]=x & 3<=i) + (a[4]=x & 4<=i);

MODULE samecount(a,b,x,i)
-- #x dans a[1..i] =? #x dans b[1..i]
VAR count_a : count(a,x,i);
    count_b : count(b,x,i);
DEFINE result:= count_a.result = count_b.result;

MODULE perm(a,b,i)
--permut(a,b,1,i,1,i)
VAR samecount1 : samecount(a,b,a[1],i);
    samecount2 : samecount(a,b,a[2],i);
    samecount3 : samecount(a,b,a[3],i);
    samecount4 : samecount(a,b,a[4],i);
DEFINE result:=  samecount1.result & samecount2.result &
                 samecount3.result & samecount4.result;

MODULE countN(a,x,i)
-- #x dans next(a[1..i])
DEFINE result := (next(a[1])=x & 1<=i) + (next(a[2])=x & 2<=i) +
                 (next(a[3])=x & 3<=i) + (next(a[4])=x & 4<=i);

MODULE samecountN(a,x,i)
-- #x dans a[1..i] =? #x dans next(a[1..i])
VAR count_a : countN(a,x,i);
    count_b : count(a,x,i);
DEFINE result:= count_a.result = count_b.result;

MODULE permN(a,i)
--permut(a,next(a),1,i,1,i)
VAR samecount1 : samecountN(a,next(a[1]),i);
    samecount2 : samecountN(a,next(a[2]),i);
    samecount3 : samecountN(a,next(a[3]),i);
    samecount4 : samecountN(a,next(a[4]),i);
DEFINE result:= samecount1.result & samecount2.result &
                samecount3.result & samecount4.result;
```

Appendix B

MPVS **Tool**

B.1 Concrete Syntax of the Languages

We display the concrete syntax of the languages. The syntax is divided in three subsections. We first specify the syntax of the declarations, then the statements and finally the assertions. Some syntactic elements of a previous subsection can be used in the following subsections.

B.1.1 Declarations

Declarations

< *declarations* >	::= **Data:** < *const_decl* > < *decl* >
	[**Auxiliary_variables:** < *decl* >
	Result_variables: < *decl* >]
< *const_decl* >	::= **const** < *id* > **:** < *integer* > **..** < *integer* > **;**
	$\mid$ **const** < *id* > **=** < *integer* > **;**
< *decl* >	::= < *var_decl* > **;**
	< *array_decl* > **;**
	< *decl* > < *decl* >
< *var_decl* >	::= **var** < *id* > **:** < *intervalle* >
	var < *id* > **: boolean**
< *array_decl* >	::= **tab** < *id* > **[1 ..** < *expr* > **] : of** < *intervalle* >
< *intervalle* >	::= < *bi* > **..** < *bs* >
< *bi* >	::= < *expr* > $\mid$ **maxint**
< *bs* >	::= < *expr* > $\mid$ **maxint**
< *op* >	::= **+** $\mid$ **−** $\mid$ **∗** $\mid$ **div** $\mid$ **mod**
< *integer* >	::= $([0-9])^+$
< *id* >	::= $[A-Z, a-z](A-Z, a-z, 0-9])^*$
< *expr* >	::= < *id* >
	$\mid$ < *integer* >
	$\mid$ < *expr* >< *op* >< *expr* >
	$\mid$ **(** < *expr* > **)**

B.1.2 The Programming Language

Instructions

< *instr* > ::= **skip**
| < *id* > **:=** (< *aexpr*> | < *bexpr* >)
| < *id* > [< *aexpr* >] **:=** < *aexpr* >
| **if** < *bexpr* > **then** < *instr* > **else** < *instr* > **end**
| **sp(** < *file* > **)**
| < *instr* > **;** < *instr* >

Expressions

< *aexpr* > ::= < *integer* >
| < *id* >
| < *id* > **[** < *aexpr* > **]**
| < *aexpr* > < *aop* > < *aexpr* >
< *bexpr* > ::= **true** | **false**
| < *id* >
| < *aexpr* > < *cop* > < *aexpr* >
| < *bexpr* > < *bop* > < *bexpr* >
| < *bnot* > < *bexpr* >
< *file* > :: = < *id* >**.in**

Operators

< *bop* > ::= **|** | **&** | **||** | **&&** | < *eop* >
< *bnot* > ::= **!**
< *aop* > ::= **+** | **−** | **∗** | **div** | **mod**
< *cop* > ::= **<** | **>** | **<=** | **>=** | < *eop* >
< *eop* > ::= **=** | **!=**

Subproblems

< *subproblem* > ::= **Data:** < *decl* >
[**Auxiliary_variables:** < *decl* >
Result_variables: < *decl* >]
precondition: < *precondition* >
postcondition: < *postcondition* >
[**invariant:** < *invariant* >
variant: < *variant* >]
< *blocs_instr* >
< *precondition* > ::= < *assert* >
< *postcondition* > ::= < *assert* >
< *invariant* > ::= < *assert* >
< *variant* > ::= < *aexpr* >
< *blocs_instr* > ::= **init:** < *instr* >
iter: < *instr* >
clot: < *instr* >
halting_condition: < *bexpr* >
|
instr: < *instr* >
Programme ::= a set of subproblems being in the same directory.

B.1.3 The Assertion Language

Assertions

```
< assert > ::= < bexpr >
             | < gaexpr > < cop > < gaexpr >
             | < assert > (< bop > | < iop >) < assert >
             | < bnot > < assert >
             | ( forall < id >:<aexpr+><=< id ><=<aexpr+>:<assert>)
             | ( exist < id >:< aexpr+ ><=< id ><=< aexpr+ >:<assert> )
             | ( exist < id >[1..< aexpr+ >] :
                     < aexpr+ ><=< id ><=< aexpr+ >:< assert> )
             | ( forall < id >[1.. < aexpr+ >] :
                     < aexpr+ ><=< id ><=< aexpr+ >:< assert> )
             | < gtab > < tbop > < gtab >
             | permut (< gtab >, <gtab>, < aexpr+ >, < aexpr+ >,
                      < aexpr+ >, < aexpr+ >)
             | unchanged (< id >)
             | unchanged (< aexpr+ >,< aexpr+ >:< gtab >)
             | initialised (< id >)
             | initialised (< aexpr+ >,< aexpr+ >:< gtab >)
```

Arithmetic expressions

```
< gaexpr >  ::= < aexpr >
              | < id >_0
              | < id >_0[ < aexpr+ > ]
              |  < aexpr >^ < aexpr >
              | ( < taop > < id > : <aexpr+> <=< id ><= < aexpr+ > :
                         < gaexpr > )
              | ( # < id > : <aexpr+> <=< id ><= <aexpr+> :
                     < assert > )
              | < gaexpr >< aop >< gaexpr >
< gtab >    ::= < id >_0 | < id >
< aexpr+ >  ::= < aexpr >
              | < id >_0
              | < id >_0[ < aexpr+ > ]
              | < aexpr+ >< aop >< aexpr+ >
```

Operators

```
< iop >     ::= => | <=>
< taop >    ::= sum | max | min
< tbop >    ::= = | != | << | <<= | >> | >>=
```

B.2 Availability of the Tool

MPVS is open source and is available on the web site *http://sourceforge.net*.

B.3 Implementation

B.3.1 Parsing and Type Checking

To make a syntactic analysis of our files, we need to define a *scanner* and a *parser*. A scanner is a program that performs lexical analysis, which means that it transforms a stream of characters into a stream of tokens. A parser is a program that performs syntax analysis. This means that a stream of tokens is analyzed and a (unique) tree structure on the tokens in this stream is computed.

To do so, we use the *Gump Scanner Generator* and the *Gump Parser Generator* modules. For both, their input consists of an Oz source with embedded scanner/parser specifications; the output is an Oz class definition. Files `AssertionScanner.ozg` and `AssertionParser.ozg` specify the tokens and the concrete syntax of our languages. The `Parser.oz` file creates the Oz classes corresponding and according to the resulting status of the parsing, he makes a type checking on this tree (eventually according to the trees of the called subproblems).

An error in the parsing or in the type checker is automatically propagated and a detailed message is given to the user.

B.3.2 Generating the Script

If no problem occurs in syntactic analysis, we can generate the script from a syntactic tree(s): in `SearchMethod.oz`, a procedure defines, from the Hoare proposition selected, the script which will be given to the search engine. To define the script, it needs to

- declare the Oz variables: it uses the `MemoriesAccess.oz` file
- generate the propagators and initialise the data structure: the responsible procedure is in the `HoareMethod.oz` file.
- define a heuristic strategy: it is defined in the `SearchMethod.oz` file

The following procedure generates the script of $\{P\}\ S\{Q\}$: `D` contains the declaration of the program variables; `SyntaxTrees` represents the set of syntactic trees; `HoareP` represents the Hoare proposition we check.

```
  fun{ScriptGenerator D SyntaxTrees HoareP}
     proc{$ Sol}
       Mp = {Dictionary.new}
       Mp0 = {Dictionary.new}
       List DS
     in
       try
          {Mem.setMemoryConst D Mp0 List}
          {FD.distribute naive List}
            %% declaration and distribution of the Oz variables
            %% corresponding to the constants
          {Mem.constraintsStoreClone Mp0 Mp}
          {Mem.setMemory He.pre D Mp0 Mp0}
          {Mem.setMemory He.p D Mp Mp0}
            %% declaration of the Oz variables
            %% corresponding to the program variables
        catch E then
          {Port.send {Access MyErrorPort} E} fail
            %% propagation of errors
        end
        Sol = {HoareM.propagGenerator D SyntaxTrees HoareP Mp Mp0 DS}
            %% generation of the propagators
            %% corresponding to the Hoare proposition HoareP
        {MyDistribution DS}
            %% definition of the distribution strategy
      end
    end
```

HoareM and Mem are the identifiers of the compilation units corresponding to the files MemoriesAccess.oz and HoareMethod.oz.

B.3.2.1 The MemoriesAccess.oz file

The following three functions are responsible for generating Oz variables from the code (according to the declaration represented in D) and linking them with the program identifier in a dictionary Mp (and Mp0). In fact D has the role of the static environment *td* of Chapter 5.

```
%%to generate the Oz variables corresponding to the program constants
proc{SetMemoryConst D Mp List}

%%to generate the Oz variables corresponding to the identifier Id
proc{SetMemoryId Id D Mp}

%%to generate the Oz variables appearing in the assertion Assert
proc{SetMemory Assert D Mp Mp0}

%%to access the Oz variable (or tuple) corresponding
%%to an identifier Id
```

```
fun{GetMemory Id Mp Mp0}
```

B.3.2.2 `HoareMethod.oz` file

To translate $\{P\}\ S\ \{Q\}$, where P is an assertion constraining the variables of the two dictionaries `Mp` and `Mp0` that have been initialised by `SetMemory` from `MemoriesAccess.oz` file, the method declaration is

```
proc{PropagGenerator D SyntaxTrees HoareP Mp Mp0 DS Sol}
```

where `D` (the *td* environment), `SyntaxTrees`(the set of syntactic trees), `HoareP`(the Hoare proposition we check) are input; `Mp`, `Mp0`, and `DS` are modifiable data structures and `Sol` is the output variable that will contain the counter-example when the script will be executed by the search engine.

B.3.2.3 `MyAssertionPropagators.oz` file

- The method `PropB` from `MyAssertionPropagators.oz` file) is deeply detailed in Chapter 6. The complete method declaration is the following:

  ```
  proc{PropB Assert Mp Mp0 Violated Feedback DS Bool Error}
  ```

  We just notice two additional outputs that are

  - `Violated` which corresponds to the subtree which is responsible to a false evaluation,
  - `Feedback` which is a record containing the kind of error in the case where the assertion is badly defined.

- It is similar for the arithmetic expressions:

  ```
  proc{PropA Gaexpr Mp Mp0 Violated Feedback DS V Neg Error}
  ```

B.3.2.4 `Auxpropagators.oz`

The generic methods like `ForEach` of Figure 6.5 are defined in a file named `AuxPropagators.oz`.

B.3.2.5 The distribution function

```
proc{MyDistribution DS}
   L1 L2 in
   {Space.waitStable} %% waits that the computational space is stable
   L = {F DS DS1}
      %% F captures in the list L the set of variables of DS
      %% that can be distributed
```

```
      %% and DS1 is the modified data structure DS
      %% from which the determined variables are removed
   case L of
      nil then skip
   []
      I|T then
      W = {FoldL
           fun{$ X J}
             Y1 = {FD.reflect.size J} Z1 ={System.nbSusps J}
             Y2 = {FD.reflect.size X} Z2 ={System.nbSusps X} in
             if Y1*Z2<Y2*Z1 orelse Z2==0
             andthen Z1==0 andthen Y1<Y2
             then J else X end
            end
          I}
         %% selection from the list of the variables to be distributed:
         %% we make a first-fail strategy
         %% taking into account
         %% the number of threads suspended by each variable
      M
   in
      M = {FD.reflect.min W}
      case {Space.choose 2} of
         1 then W = M  {MyDistribution DS1}
      []
         2 then W \=: M  {MyDistribution DS1}
      end
      %% realisation of the tree
    end
end
```

B.3.3 The GUI

The graphical interface is described in file `FenetrePrincipale.oz`, it has been implemented using the Tk module. The module `QTK` is more recent and would more easily to be used.
Anyway, the functionalities are simple:

- functionalities to open, close and save files,
- functionalities to go from one file to another,
- functionalities to hightlight some text in the opened files,
- functionalities to display a counter-example in a table,
- functionalities to check the correctness of a Hoare proposition of a file:

  `Parser` is used to generate the syntactic tree

  `SearchMethod` is used to generate the script and call the search engine.

According to the counter-example found, a message is displayed in the feedback area and the record containing the badly defined or violated subexpression allows us to underline it in the text area (thanks to the position in the file that we keep in the syntactic tree), as well as the involved sequence of statements.

Of course if, no counter-example is found, a message attesting the correctness of the Hoare proposition is written in the feedback area.

B.4 Testing Students' Solutions to the Subarray with Biggest Sum Problem

This Appendix concerns the first project given to the student in our second experimentation (in 2007): to elaborate in $O(m^2n)$, a Java algorithm finding the greatest value corresponding to the sum of a non-empty subarray of a two-dimensional array $a[1..m][1..n]$. In this Appendix, the reader has details about the tests automatically generated and the students performances. We present here five tests with an array content and the expected result.

```
Test data and correct results
------------------------------
                          0             1             2
              +-------------+-------------+-------------+
            0 |  2147483647 |           1 | -2147483648 |
              +-------------+-------------+-------------+
            1 | -2147483647 |          -1 | -2147483647 |
              +-------------+-------------+-------------+
            2 |  2147483647 |          -1 | -2147483647 |
              +-------------+-------------+-------------+
A subarray of maximal sum is a[0:0][0:1].
The sum is : 2147483648

                          0             1
              +-------------+-------------+
            0 |  2147483647 |           1 |
              +-------------+-------------+
                          0
              +-------------+
            0 | -1207558202 |
              +-------------+
A subarray of maximal sum is a[0:0][0:0].
The sum is : -1207558202

                          0
              +-------------+
            0 | -1207558202 |
              +-------------+
                         0            1            2            3            4
              +------------+------------+------------+------------+------------+
            0 | -624492714 |  578023413 |  -61948189 | -739099171 |  334225298 |
              +------------+------------+------------+------------+------------+
A subarray of maximal sum is a[0:0][1:1].
The sum is : 578023413

                         1
              +------------+
            0 |  578023413 |
              +------------+

                           0
              +-------------+
            0 | -1495484563 |
              +-------------+
            1 | -1531128095 |
              +-------------+
            2 |   734713349 |
              +-------------+
            3 |   328665974 |
              +-------------+
            4 |  -877842511 |
              +-------------+
            5 | -1197603801 |
              +-------------+
            6 |  -409777642 |
              +-------------+
            7 |   -42586036 |
              +-------------+
            8 |  -433009038 |
              +-------------+
            9 |  -689769219 |
              +-------------+
           10 |   -15590802 |
              +-------------+
           11 |  -489849945 |
              +-------------+
           12 |   979004627 |
              +-------------+
           13 |  -789803059 |
              +-------------+
           14 |   623199939 |
              +-------------+

A subarray of maximal sum is  a[2:3][0:0].
The sum is : 1063379323

                           0
              +-------------+
            2 |   734713349 |
              +-------------+
            3 |   328665974 |
              +-------------+


                           0             1             2             3
              +-------------+-------------+-------------+-------------+
            0 |   -96455806 |   472732538 |   488705406 | -1325001031 |
              +-------------+-------------+-------------+-------------+
            1 |  1016899380 |  -388234579 | -1113064636 |   951334362 |
              +-------------+-------------+-------------+-------------+
            2 | -1268123216 |   601153015 |   237908767 |   803365760 |
              +-------------+-------------+-------------+-------------+
            3 | -1795984665 |  -549764834 | -1823124583 | -1938761833 |
              +-------------+-------------+-------------+-------------+

A subarray of maximal sum is a[1:2][3:3].
The sum is : 1754700122

                          3
              +-------------+
            1 |   951334362 |
              +-------------+
            2 |   803365760 |
              +-------------+
```

Here is the tests evaluation for each student (identified by no icampus). Ti are the tests, showing 0 in case of failure and 1 in case of success.

```
no iCampus |T1|T2|T3|T4|T5|
-----------|--|--|--|--|--|
         1 | 0| 0| 0| 0| 0|
         2 | 1| 1| 1| 1| 1|
         3 | 0| 1| 1| 1| 0|
         4 | 1| 0| 1| 0| 0|
         5 | 1| 1| 1| 1| 0|
         6 | 1| 1| 1| 0| 0|
         7 | 1| 1| 1| 1| 1|
         8 | 0| 0| 0| 0| 0|
         9 | 1| 1| 1| 0| 0|
        10 | 0| 1| 1| 0| 1|
        11 | 0| 0| 0| 0| 0|
        12 | 1| 1| 1| 0| 0|
        13 | 1| 1| 1| 0| 0|
        14 | 0| 0| 0| 0| 0|
        15 | 1| 1| 1| 1| 0|
        16 | 0| 0| 0| 0| 0|
        17 | 1| 1| 1| 1| 1|
        18 | 1| 1| 1| 1| 1|
        19 | 1| 1| 1| 1| 1|
        20 | 0| 0| 0| 0| 0|
        21 | 0| 0| 0| 0| 0|
        22 | 0| 0| 0| 0| 0|
        23 | 0| 0| 0| 0| 0|
        24 | 0| 0| 0| 0| 0|
        25 | 1| 1| 1| 0| 0|
        26 |  1|  1|  1|  0|  0|
        27 |  1|  0|  1|  1|  1|
        28 |  1|  1|  1|  0|  0|
        29 |  1|  1|  1|  1|  1|
        30 |  0|  1|  1|  0|  1|
        31 |  1|  1|  1|  1|  1|
        32 |  1|  1|  1|  1|  0|
        33 |  0|  0|  0|  0|  0|
        34 |  0|  0|  0|  0|  0|
        35 |  1|  1|  1|  0|  0|
        36 |  0|  1|  1|  1|  1|
        37 |  1|  1|  1|  0|  0|
        38 |  0|  0|  0|  0|  0|
        39 |  0|  1|  1|  1|  0|
        40 |  1|  1|  1|  0|  0|
        41 |  1|  1|  1|  0|  0|
        42 |  1|  0|  1|  0|  0|
        43 |  1|  1|  1|  1|  1|
        44 |  0|  0|  0|  0|  0|
        45 |  0|  1|  1|  0|  1|
        46 |  1|  1|  1|  1|  1|
        47 |  1|  1|  1|  1|  1|
        48 |  0|  0|  0|  0|  0|
        49 |  0|  0|  0|  0|  0|
        50 |  0|  0|  0|  0|  0|
        51 |  0|  0|  0|  0|  0|
        52 |  1|  1|  1|  0|  0|
```

B.5 Solutions to the Exercises Presented in Chapter 7

In this section, we provide the solutions for the exercises given to the students during the tool experimentations.

```
Data: const n <= 5 ;
      const minv = 0 ;
      const maxv = 2 ;
      tab   a : array [1..n] of minv .. maxv ;
      var   x : minv .. maxv ;

Auxiliary_variables:
      var   i : 0 .. maxint ;

Result_variables:
      var   present : boolean ;

Precondition: initialised(x) & initialised(1, n : a)

Postcondition: unchanged(x) & unchanged(1, n : a) &&
      (present <=> (exist i : 1 <= i <= n : a[i] = x))

Invariant: unchanged(x) & unchanged(1, n : a) & 0 <= i & i <= n
           && (forall j : 1 <= j <= i : a[j] != x)

Init: i := 0

Iter: i := i +1

Clot: present:= !(i=n)

Halting_condition: i = n || a[i +1]=x

Variant: n - i
```

Figure B.1: The sequential search exercise

```
Data: const n <= 5 ;
  const noir = 7 ;
  const jaune = 8 ;
  const rouge = 9 ;
  tab a : array[1 .. n] of noir .. rouge ;

Auxiliary_variables:
  var in : 0 .. maxint ;
  var ij : 0 .. maxint ;
  var ir : 0 .. maxint ;
  var x : noir .. rouge ;

Precondition: initialised(1, n : a)

Postcondition: permut(a, a_0, 1, n, 1, n) &
  (exist i : 0 <= i <= n :
     (exist j : i <= j <= n :
        (forall kn : 1     <= kn <= i : a[kn] = noir)
      & (forall kj : i + 1 <= kj <= j : a[kj] = jaune)
      & (forall kr : j + 1 <= kr <= n : a[kr] = rouge)
  ))

Invariant: 0 <= in & in <= ij & ij <= ir & ir <= n &&
  permut(a, a_0, 1, ir, 1, ir) & unchanged(ir + 1, n : a) &
  (forall kn : 1 <= kn <= in : a[kn] = noir) &
  (forall kj : in + 1 <= kj <= ij : a[kj] = jaune) &
  (forall kr : ij + 1 <= kr <= ir : a[kr] = rouge)

Init: in := 0; ij := 0; ir := 0

Iter: x := a[ir + 1];
      ir := ir + 1;
      if a[ir]= rouge then skip
      else ij := ij+1;
           a[ir] := a[ij];
           if x = jaune then a[ij] := x
           else  in := in+ 1;
                 a[ij] := a[in];
                 a[in] := x
           end
       end

Clot: skip

Halting_condition: ir = n

Variant:  n - ir
```

Figure B.2: The Belgian flag exercise

```
Data:
  const m :1..3 ;
  const n : 1..3 ;
  const minv = 45 ;
  const maxv = 51 ;
  tab a : array [1 .. m] of minv .. maxv ;
  tab b : array [1 .. n] of minv .. maxv ;

Auxiliary_variables: var ia : 0..maxint ;
                     var ib : 0..maxint ;

Result_variables:
  var k : 0 .. maxint ;

Precondition:
  (forall i : 1 <= i <= m - 1 : a[i] < a[i + 1]) &
  (forall i : 1 <= i <= n - 1 : b[i] < b[i + 1])

Postcondition:
  unchanged(1, m : a) & unchanged(1, n : b) &
  k = (# v : minv <= v <= maxv :
             (exist i : 1 <= i <= m : a[i] = v) &
             (exist j : 1 <= j <= n : b[j] = v)
       )

Invariant:
 unchanged(1, m : a) & unchanged(1, n : b) &
  1 <= ia & ia <= m + 1 & 1 <= ib & ib <= n + 1
 &&
  k = (# v : minv <= v <= maxv :
             (exist i : 1 <= i <= ia -1 : a[i] = v) &
             (exist j : 1 <= j <= ib - 1 : b[j] = v))
  &
  ((ia = m+1 | ib = 1)|| a[ia] > b[ib-1] )
  &
  ((ib = n+1 | ia = 1)|| b[ib] > a[ia-1])


Init:  ia := 1 ; ib := 1 ; k := 0

Iter:  if a[ia] < b[ib] then ia := ia + 1
       else if a[ia] > b[ib] then ib := ib + 1
       else ia := ia + 1 ; ib := ib + 1 ; k := k + 1
       end
end

Clot: skip

Halting_condition: ia= m + 1 | ib = n + 1

Variant: m + n + 2 - ia - ib
```

Figure B.3: The number of values common to two strictly sorted arrays

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:
  var   g : 0 .. n ;
  var   d : 0 .. n ;

Result_variables:
  var  nv : 0 .. n ;

Precondition:
  initialised(v) &
  (forall i : 2 <= i <= n : a[i - 1] <= a[i])

Postcondition:
  unchanged(v) & unchanged(1, n : a) &
  nv = (# i : 1 <= i <= n : a[i] = v)

Instr:
  sp(dichoG.in) ; sp(dichoD.in) ; nv := d - g
```

Figure B.4: The number of occurrences of the value `v` in the array `a` in $log(n)$ time

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:
  var   i : 0 .. n ;
  var   s : 0 .. n ;
  var   m : 0 .. n ;


Result_variables:
  var   g : 0 .. n ;

Precondition: initialised(v) &
  (forall j : 2 <= j <= n : a[j - 1] <= a[j])

Postcondition:
  unchanged(v) & unchanged(1, n : a) &
  0 <= g & g <= n &&
  (forall j : 1 <= j <= g : a[j] < v) &
  (forall j : g + 1 <= j <= n : v <= a[j])

Invariant:
  unchanged(v) & unchanged(1, n : a) &
  0 <= i & i <= s & s <= n &&
  (forall j : 1 <= j <= i : a[j] < v) &
  (forall j : s + 1 <= j <= n : v <= a[j])

Init:
  i := 0 ; s := n

Iter:
  m := (i + s + 1) div 2 ;
  if a[m] < v then i := m else s := m - 1 end

Clot:
  g := i

Halting_condition:
  i = s

Variant:
  s - i
```

Figure B.5: Subproblem SP1 using a binary search to compute the number of occurrences of `v` in `a` in $log(n)$ time

```
Data:
  const n <= 5 ;
  const minv = 97 ;
  const maxv = 99 ;
  tab   a : array [1 .. n] of minv .. maxv ;
  var   v : minv .. maxv ;

Auxiliary_variables:
  var   i : 0 .. n ;
  var   s : 0 .. n ;
  var   m : 0 .. n ;


Result_variables:
  var   d : 0 .. n ;

Precondition: initialised(v) &
  (forall j : 2 <= j <= n : a[j - 1] <= a[j])

Postcondition:
  unchanged(v) & unchanged(1, n : a) &
  0 <= d & d <= n &&
  (forall j : 1 <= j <= d : a[j] <= v) &
  (forall j : d + 1 <= j <= n : v < a[j])

Invariant:
  unchanged(v) & unchanged(1, n : a) &
  0 <= i & i <= s & s <= n &&
  (forall j : 1 <= j <= i : a[j] <= v) &
  (forall j : s + 1 <= j <= n : v < a[j])

Init:
  i := 0 ; s := n

Iter:
  m := (i + s + 1) div 2 ;
  if a[m] <= v then i := m else s := m - 1 end

Clot:
  d := i

Halting_condition:
  i = s

Variant:
  s - i
```

Figure B.6: Subproblem SP2 using a binary search to compute the number of occurrences of `v` in `a` in $log(n)$ time

Bibliography

[1] P. Amay. Correctness by Construction: Better can also be Cheaper. *CrossTalk*, pages 24–28, 2002.

[2] J. Arsac. Vous avez dit algorithmique? In *Actes du deuxième colloque francophone sur la didactique de l'informatique*, 1990.

[3] J. Barnes. *High Integrity Software: The Spark Approach to Safety and Security.* Addison-Wesley, 2003.

[4] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-Order Processor Verification. In *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*, volume LNCS 1522, pages 369–386. Springer-Verlag, 1998.

[5] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - A second generation of a Java model checker. In *Proc. of the Workshop on Advances in Verification*, Chicago, Illinois, July 2000.

[6] J. v. C.-B. Breunesse and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In C. R. H. Kirchner, editor, *Algebraic Methodology and Software Technology*, volume LNCS 2422, pages 304–318. Springer-Verlag, 2002.

[7] L. Burdy, Y. Cheon, D. R. Cok, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML and applications. In *Eighth International Workshop on Formal methods for Industrial Critical System (FMICS 03)*, volume 80, pages 73–89, Elsevier, 2003. Electronic Notes in theoretical Computer Sciences.

[8] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Proc. of Formal Methods Europe (FME 2002)*, volume 2391, pages 272–289. LNCS Springer-Verlag, 2002.

[9] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, J. Reeseand, and al. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[10] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002- Object-Oriented Programming, 16th European Conference*, volume LNCS 2374, Spain, 2002. Springer-Verlag.

[11] E. Clarcke, D. Kroening, and F. Lerna. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume LNCS 2988, pages 168–176. Springer, 2004.

[12] D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proc. of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume LNCS 3362, pages 108–128. Springer, 2004.

[13] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Sofware Engeneering*, pages 439–448. ACM, 2000.

[14] M. Derroitte and B. Le Charlier. Un système d'aide à l'enseignement d'une méthode de programmation. In *Actes du premier colloque francophone sur la didactique de l'informatique*, 1989.

[15] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical report, Compaq System Research Center, 2002.

[16] D.L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J.B. Saxe. Extended Static Checking. Technical Report Research Report 159, Compaq Systems Research Center, 1998.

[17] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.

[18] I. Dony and B. Le Charlier. Finding errors with Oz. In *Techniques for implementing constraint programming system, Workshop held in conj. with CP2002, 8th International Conference on practice of constraint programming*, Ithaca, 2002.

[19] I. Dony and B. Le Charlier. Why don't we Simply Use a Model Checker. In *Third International Workshop on Constraints in Formal Verification*, 2003.

[20] I. Dony and B. Le Charlier. A program Verification System based on Oz. In *Proc. second International Conference MOZ 2004*, volume LNCS 3389, Charleroi, Belgium, 2004. Springer.

[21] I. Dony and B. Le Charlier. A Tool for Helping Teach a Programming Method. In *The Eleventh Annual Conference on Innovation and Technology in Computer Science Education*, volume 38, pages 212–216, University of Bologna, Italy, 2006. ACM Press.

[22] I. Dony and B. Le Charlier. Why don't we Simply Use a Model Checker (or a tool based on general theorem proving)? *submitted for a special issue of the Journal on Satisfiability, Boolean Modeling and Computation (JSAT) on the topic of application of constraints to formal verification (CFV)*, 2007.

[23] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002*, volume 37, pages 234–245, New York, 2002. Conference on Programming Language Design and Implementation(PLDI'02), ACM Press.

[24] R.W. Floyd. Assigning meaning to programs. In *Proc. of Symposia in Applied Mathematics*, volume 19, pages 19–32. Mathematical Society, 1967.

[25] N. Gehani. *Ada: An Advanced Introduction*. Prentice-Hall, Engelwood Cliffs, Nj, 1983.

[26] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[27] T.A. Henzinger, R. Jhala, R. Majumbar, and G. Sutre. Lazy abstraction. In *In proc. of the 29th Annual Symposium on Principles of Programming Language*, pages 58–70. ACM Press, 2002.

[28] C.A.R. Hoare. An axiomatic approach to computer science. *Communications of the ACM*, 12, 1969.

[29] C.A.R. Hoare. An axiomatic definition of semantics. *Communications of the ACM*, 12(10), 1969.

[30] C.A.R. Hoare. Procedures and parametres: an axiomatic approach. *In Symposium on Semantics of Programming Languages*, pages 102–116, 1971.

[31] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[32] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335–355, 1973.

[33] G. Holzmann. The Spin model checker. *IEEE Transactions on Sofware Engeneering*, 23(5):279–295, May 1997.

[34] Bergeretti J.-F. and B. A. Carré. Information Flow and Data-Flow Analysis of while-Programs. In ACM, editor, *ACM Transactions on Programming Languages and Systems*, volume 7, pages 37–61, New York, 1985.

[35] J. R. Kiniry. The Logics and Calculi of ESC/Java2. Technical report, 2004.

[36] K. Lau. A beginner's Course on Reasoning About Imperative Programs. In *Proc. Symposium on Teaching Formal Methods*, volume LNCS 3294, Ghent, Belgium, 2004. Springer-Verlag.

[37] B. Le Charlier. *Introduction à la programmation*. Librairie des Sciences des FUNDP, Namur, 1999.

[38] B. Le Charlier. Introduction à l'Algorithmique et à la Programmation. http://www.icampus.info.ucl.ac.be/SINF1160/, 2007.

[39] B. Le Charlier. Méthodes de Conception de Programmes. http://www.icampus.info.ucl.ac.be/INGI2122/, 2007.

[40] B. Le Charlier and P. Flener. Specifications are necessarily informal or: some more myths of formal methods. *The journal of Systems and Software*, March, 1998.

[41] G. T. Leavens and Y Cheon. Design by Contract with JML. In *MOVEP'04 6th school on MOdeling and VErifying parallel Processes*, Université Libre de Bruxelles, Belgium, 2004.

[42] J. P. Marques-Silva and K. A. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proc. of International Conference on Computer-Aided Design*, pages 220–227, Santa Clara, California, U.S.A., 1996.

[43] K. L. McMillan. The SMV model checking system. http://www.cis.ksu.edu/santos/smv-doc/.

[44] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[45] MOBIUS. Mobility ubiquity security. http://mobius.inria.fr, 2007.

[46] P. Naur. Proofs of algorithms by general snapshots. *BIT 6*, pages 310–316, 1969.

[47] G. Nelson. Combining satisfiability procedures by equality-sharing. In W. W. Bledsoe and D. W. Loveland, editors, *In Automatic Theorem Proving*, pages 201–211. American Mathematical Society, 1983.

[48] A. S. Ruocco. Experiences using Spark in an Undergraduate CS Course. In *Proc. of the ACM SIGAda Annual International Conference*, volume 25(4), pages 37–40. ACM Press, 2005.

[49] C. Schulte. *Programming Constraint Services*, volume 2302 of Lecture Notes in Artificial Intellingence. Springer-Verlag, Berlin, Germany, 2002.

[50] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* The Mit Press, 2004.