

## Chapter 8

# Computer Programs

*Der liebe Gott gibt uns die Logik des Algorithmus. Der Teufel fügt den Programmierer hinzu und schon ist das irdische Gleichgewicht wieder hergestellt. (God gives us the logic of the algorithm. The devil adds the programmer and already is the earthly balance restored.) — Heinz Zemanek*

The semantic formalisms presented in Chapter 7 define the meaning of programming languages and thus implicitly also of computer programs written in these languages. However, they are too low-level to be of practical use when our actual goal is to verify the correctness of programs with respect to given specifications. In this chapter, we are going to discuss various (closely related) calculi that allow us to reason about programs on the high level. These calculi can be subsumed under the term *axiomatic semantics*, because they are based on axioms and inference rules that describe how we can derive expected program properties. The soundness of these calculi can be shown from the underlying denotational semantics of the programming language.

Our presentation starts in Section 8.1 with a discussion of *problem specifications* that serve as *program contracts*; we subsequently develop programs that satisfy these contracts, i.e., solve the specified problems. Then Section 8.2 introduces the *Hoare Calculus*, the father of axiomatic semantics; this calculus can be viewed as an abstract procedure for the generation of *verification conditions*, i.e., logical formulas whose validity implies the correctness of a program. Section 8.3 further elaborates the theory of verification condition generation to the calculi of *weakest preconditions*, *strongest postconditions*, and *commands as relations*, which are more suitable for implementation in automated verification systems. While the previous presentation have only considered a program's *partial correctness* (the program does not produce a wrong result), Section 8.4 deals with its *total correctness* (the program indeed produces a result, i.e., it neither aborts prematurely nor does it fail to terminate).

So far the chapter has focused mainly on the “mechanics” of program verification; in Section 8.5 we also discuss its “pragmatics”. In particular, we investigate by a series of examples strategies for the elaboration of “meta-information” in the form of *loop invariants* and *termination measures* which the human verifier has to provide to make a verification succeed. Also the discussion so far has centered entirely on commands, i.e., on “verification on the small”. As a preparation for “verification in the large”, Section 8.6 addresses the *refinement* of commands. Section 8.7 builds upon the insights gained there to finally discuss the contract-based *modular verification* of programs composed of procedures.

## 8.2 Verifying Programs

As a starting point for program verification, we will in this and in the subsequent sections investigate various (closely related) formalisms to verify the correctness of the commands that make up a program.

### The Hoare Calculus

We begin with a formalism for program reasoning that was invented in 1969 by the British computer scientists C.A.R. Hoare; this formalism was later called “Hoare Calculus” or “axiomatic program semantics” (in contrast to denotational semantics or operational semantics, axiomatic semantics describes how we can reason about the externally observable effect of a command without describing its internal mechanism). This calculus is the most popular one when it comes to manually proving the correctness of programs (automated or semi-automated program verifiers typically implement somewhat more advanced calculi; some of these will be discussed later).

The core judgement of the Hoare calculus is the “Hoare triple”

$$\{P\} C \{Q\}$$

where  $C$  is a command and  $P$  and  $Q$  are formulas. For the moment, our informal interpretation of this judgement is the following:

If the execution of command  $C$  starts in a state in which the precondition  $P$  holds, every normal termination of the command yields a state in which the postcondition  $Q$  holds.

The pair  $\langle P, Q \rangle$  thus represents a “specification” to be implemented by command  $C$ . Please note that this specification only claims “partial correctness” of the command execution; it does not rule out that the command aborts or runs forever (we will deal with these issues later).

The Hoare calculus is an inference system for deriving Hoare triples; it is sound in the sense that by its rules we can only derive Hoare triples that are true with respect to the partial correctness interpretation given above.

### Variable Substitutions

Before presenting the Hoare calculus, we introduce a technical concept that is applied in some rules of the calculus.

**Proposition 8.1 (Variable Substitution).** Let  $E$  be a phrase (formula or term),  $V$  a variable and  $T$  a term. The  $E[T/V]$  is defined as the phrase

$$E[T/V] := \text{let } V = T \text{ in } E$$

whose value is the value of  $E$  when  $V$  is interpreted as the value of  $T$  (see Definition 2.16).

Alternatively (and more intuitively),  $E[T/V]$  can be understood as phrase  $E$  where every free occurrence of variable  $V$  has been syntactically replaced by term  $T$ . For instance, for  $F := x > z$ , formula  $F[x + 1/x]$  may be understood either as let  $x = x + 1$  in  $x > z$  or as  $x + 1 > z$ . While the alternative understanding of  $E[T/V]$  by syntactic substitution seems simpler, it actually requires the appropriate renaming of all quantified variables in  $E$  such that they do not coincide with any free variables in  $T$ . Since this makes the general formalization more complicated, we prefer the simple definition given above (but will use the easier to understand interpretation by syntactic substitution, if  $E$  does not contain any quantifiers).

### The Generation of Verification Conditions

The Hoare calculus is depicted in Figure 8.1. Before we dive into the details of the individual inference rules of the calculus, we demonstrate the big picture of their application. For instance, we may use these rules to derive the Hoare triple

$$\{\text{true}\} m := x; \text{ if } m < y \text{ then } m := y \{Q\}$$

for postcondition  $Q \Leftrightarrow (m = x \vee m = y) \wedge (m \leq x \wedge m \leq y)$ . This triple expresses the correctness of a command that computes the maximum  $m$  of two numbers  $x$  and  $y$ ; it is derived by the following inference tree:

$$\frac{\frac{\frac{\models \boxed{\text{true} \Rightarrow x = x} \quad \{x = x\} m := x \{m = x\}}{\{\text{true}\} m := x \{m = x\}} \quad T \quad \models \boxed{m = x \wedge \neg(m < y) \Rightarrow Q}}{\{m = x\} \text{ if } m < y \text{ then } m := y \{Q\}}}{\{\text{true}\} m := x; \text{ if } m < y \text{ then } m := y \{Q\}}$$

Here  $T$  represents the following subtree:

$$\frac{\models \boxed{m = x \wedge m < y \Rightarrow Q[y/m]} \quad \{Q[y/m]\} m := y \{Q\}}{\{m = x \wedge m < y\} m := y \{Q\}}$$

The application of these rules is mostly (but not completely) mechanically guided by the structure of the command. There is one inference rule for every command; in above example, the rules for command sequences, assignments, and one-sided conditional were applied. The rules for the compound commands (command sequences and one-sided conditional) reduce the problem of deriving the Hoare triple for a compound

**Judgement**

$$\{\_\} \sqcup \{\_\} \subseteq \text{Formula} \times \text{Command} \times \text{Formula}$$

**Rules for  $\{P\} C \{Q\}$ :**

$$\frac{\models P \Rightarrow P' \quad \{P'\} C \{Q'\} \quad \models Q' \Rightarrow Q}{\{P\} C \{Q\}}$$

$$\frac{\{Q[T/V]\} V := T \{Q\}}{\{P\} C \{Q[V_0/V]\}}$$

$$\frac{V, V_0, V_1 \text{ distinct} \quad V_0, V_1 \text{ not in } P, C, Q}{\{(\forall V_1: S. P[V_1/V])[V/V_0]\} \text{ var } V: S; C \{Q\}}$$

$$\frac{\{P\} C_1 \{R\} \quad \{R\} C_2 \{Q\}}{\{P\} C_1; C_2 \{Q\}}$$

$$\frac{\{P \wedge F\} C_1 \{Q\} \quad \{P \wedge \neg F\} C_2 \{Q\}}{\{P\} \text{ if } F \text{ then } C_1 \text{ else } C_2 \{Q\}} \quad \frac{\{P \wedge F\} C \{Q\} \quad \models P \wedge \neg F \Rightarrow Q}{\{P\} \text{ if } F \text{ then } C \{Q\}}$$

$$\frac{\models P \Rightarrow I \quad \{I \wedge F\} C \{I\} \quad \models I \wedge \neg F \Rightarrow Q}{\{P\} \text{ while } F \text{ do } C \{Q\}}$$

**Fig. 8.1** The Hoare Calculus (Partial Correctness)

command to the problem of deriving Hoare triples for its subcommands; this process terminates with the rule for the for assignment command, which is atomic, i.e., it has no subcommand. The assignment rule therefore has no premise, i.e., it is an axiom. The leaves of the tree are either Hoare triples that are instances of the assignment axiom or they are of the form  $\models F$ . The later represents the validity of formula  $F$ , which has to be established by logical reasoning; we call  $F$  a *verification condition* (in above derivation the verification conditions are marked with frames). The Hoare calculus thus implicitly describes a *verification condition generator*, a syntax-guided procedure that derives from a Hoare triple a set of verification conditions: these conditions are formulas in first order logic whose combined validity implies the correctness of the triple.

Figure 8.2 makes this procedure explicit by the definition of a function  $v\text{cg}$  that takes a Hoare triple and returns the set of verification conditions determined by the Hoare calculus; the verification condition generated for each kind of command is directly derived from the corresponding inference rule. Since the inference rules for the assignment command and for the variable declaration command do not have arbitrary preconditions, the function  $v\text{cg}$  applies the knowledge of the first rule of the calculus to generate a verification condition that demands that the given precondition  $P$  implies the precondition derived by the inference rule.

Considering the example elaborated above, the verification condition generator therefore generates the following result:

$vcg: Formula \times Command \times Formula \rightarrow Set(Formula)$

$$\begin{aligned}
 vcg(\{P\} V := T \{Q\}) &:= \{P \Rightarrow Q[T/V]\} \\
 vcg(\{P\} \text{ var } V: S; C \{Q\}) &:= \text{let } P_1 = \text{choose } F. F \in Formula \text{ in} \\
 &\quad vcg(\{P_1\} C \{Q[V_0/V]\}) \cup \\
 &\quad \{P \Rightarrow (\forall V_1: S. P_1[V_1/V])[V/V_0]\} \\
 vcg(\{P\} C_1; C_2 \{Q\}) &:= \text{let } R = \text{choose } F. F \in Formula \text{ in} \\
 &\quad vcg(\{P\} C_1 \{R\}) \cup vcg(\{R\} C_2 \{Q\}) \\
 vcg(\{P\} \text{ if } F \text{ then } C_1 \text{ else } C_2 \{Q\}) &:= vcg(\{P \wedge F\} C_1 \{Q\}) \cup \\
 &\quad vcg(\{P \wedge \neg F\} C_2 \{Q\}) \\
 vcg(\{P\} \text{ if } F \text{ then } C \{Q\}) &:= vcg(\{P \wedge F\} C_1 \{Q\}) \cup \{P \wedge \neg F \Rightarrow Q\} \\
 vcg(\{P\} \text{ while } F \text{ do } C \{Q\}) &:= \text{let } I = \text{choose } F. F \in Formula \text{ in} \\
 &\quad \{(P \Rightarrow I), (I \wedge \neg F \Rightarrow Q)\} \cup vcg(\{I \wedge F\} C \{I\})
 \end{aligned}$$

**Fig. 8.2** The Hoare Calculus as a Verification Condition Generator

$$\begin{aligned}
 vcg(\{\text{true}\} m := x; \text{ if } m < y \text{ then } m := y \{Q\}) &:= \\
 &\quad \left\{ \boxed{\text{true} \Rightarrow x = x}, \boxed{m = x \wedge m < y \Rightarrow Q[y/m]}, \boxed{m = x \wedge \neg m < y \Rightarrow Q} \right\}
 \end{aligned}$$

Thus, to verify the correctness of the given Hoare triple, it suffices to show the validity of the three generated verification conditions.

Program verification thus consists of two essentially independent parts:

- the syntax-guided derivation of verification conditions (e.g., by the Hoare calculus),
- the establishment of the validity of the verification conditions (e.g., by proving the formulas with the inference rules of first order logic).

Most rules of the Hoare calculus have a form that allows a mechanization of the first part, but this is not true for all rules: as the definition of the verification condition generator  $vcg$  makes very explicit, the three rules for a variable declaration, a command sequence, and a loop require choices of formulas that are not directly derived from the input; indeed, only for suitable choices, the derived verification conditions are valid and the verification can succeed. We will see later how other calculi replace most of these choices by systematic calculations, but nevertheless for the verification of a loop command always the choice of a *loop invariant*  $I$  is required. This choice is the main creative step that has to be performed by a clever reasoner (human or machine); we will discuss this creative aspect later. To subsequently establish the validity of the generated verification conditions is a separate process that can be also performed by a combination of human intelligence and sophisticated automation.

In a manual verification, the application of the Hoare rules to derive verification conditions is rarely given in the form of detailed inference trees but are typically indicated by a semi-formal text such as the following one:

We verify  $\{\text{true}\} m := x; \text{ if } m < y \text{ then } m := y \{Q\}$ . Clearly we have  $\{\text{true}\} m := x \{m = x\}$ . Thus it suffices to show  $\{m = x\} \text{ if } m < y \text{ then } m := y \{Q\}$ :

- We verify  $\{m = x \wedge m < y\} m := y \{Q\}$ . For this we assume  $m = x$  and  $m < y$  and show  $Q[y/m]$ . . .
- We show  $m = x \wedge \neg(m < y) \Rightarrow Q$ , i.e., we assume  $m = x$  and  $\neg(m < y)$  and show  $Q$ . . .

Also we will use in the following such descriptions from which the reader has to deduce the corresponding inference trees.

### The Rules of the Hoare Calculus

After this first exposition of the Hoare calculus, we go into the details of its rules which are depicted in Figure 8.1 and illustrate them by small examples.

The first rule of the Hoare calculus is “generic” (command-independent).

- **Weakening/Strengthening:**

$$\frac{\models P \Rightarrow P' \quad \{P'\} C \{Q'\} \quad \models Q' \Rightarrow Q}{\{P\} C \{Q\}}$$

This rule states that the problem of deriving a Hoare triple  $\{P\} C \{Q\}$  can be reduced to the problem of deriving another Hoare triple  $\{P'\} C \{Q'\}$  provided that

- the new precondition  $P'$  is not stronger than the original precondition  $P$  (i.e.,  $P'$  is implied by  $P$ ), and
- the new postcondition  $Q'$  is not weaker than the original postcondition  $Q$  (i.e.,  $Q'$  implies  $Q$ ).

In a nutshell, this rule states that, in the course of a correctness proof, we may resort to the following principle:

Preconditions may be weakened and postconditions may be strengthened.

**Example 8.5.** According to the following rule instance

$$\frac{\models x > 0 \Rightarrow x \geq 0 \quad \{x \geq 0\} C \{y = 1 + x\} \quad \models y = 1 + x \Rightarrow y \neq x}{\{x > 0\} C \{y \neq x\}}$$

the problem of deriving the triple  $H_1 = \{x > 0\} C \{y \neq x\}$  may be reduced to the problem of deriving  $H_2 = \{x \geq 0\} C \{y = 1 + x\}$ , because  $x > 0$  implies  $x \geq 0$  and  $y = 1 + x$  implies  $y \neq x$ . Thus any command  $C$  that satisfies the interpretation of  $H_2$  also satisfies the interpretation of  $H_1$ .  $\square$

To illustrate the soundness of the reasoning applied in above example, we may assume that  $C$  satisfies the interpretation of  $H_2$ . To show that then  $C$  also satisfies the interpretation of  $H_1$ , we have to show that if  $C$  starts execution in a state

# Program Reasoning in RISCAL and the RISC ProgramExplorer

There are numerous software systems that support the verification of programs in various languages, either by checking their executions in finite models or by proving verification conditions. We will present two such systems:

- RISCAL [82,97] is a specification language and associated software system for modeling algorithms and specifying their behavior in first-order logic. The type system of RISCAL ensures that the domains of all programs and formulas are finite; this allows a model checker to fully automatically check in small models all possible program executions with respect to their contracts and other annotations (thus ensuring that invariants are not too strong) but also the validity of verification conditions (thus ensuring that invariants are not too weak). Thus errors may be detected quickly before attempting a proof-based verification for models of arbitrary size.
- The RISC ProgramExplorer [95,96] is a system for the verification of programs written in a subset of Java where the contracts of Java methods are specified in first-order logic. The system translates the bodies of procedures to state relations which are appropriately simplified and presented to the user such that she may inspect the relations with respect to their adequacy to satisfy the specified contracts. If no apparent errors are detected, she may discharge the verification conditions generated from the state relations with the help of the previously presented RISC ProofNavigator [93,94] as a semi-automatic proof assistant.

The specifications used in the following presentations can be downloaded from the URLs

<https://www.risc.jku.at/people/schreine/TP/software/prog/prog.txt>  
<https://www.risc.jku.at/people/schreine/TP/software/prog/Prog.java>

and loaded by executing from the command line the following commands:

```
RISCAL prog.txt &  
ProgramExplorer & (in an empty directory containing Prog.java)
```

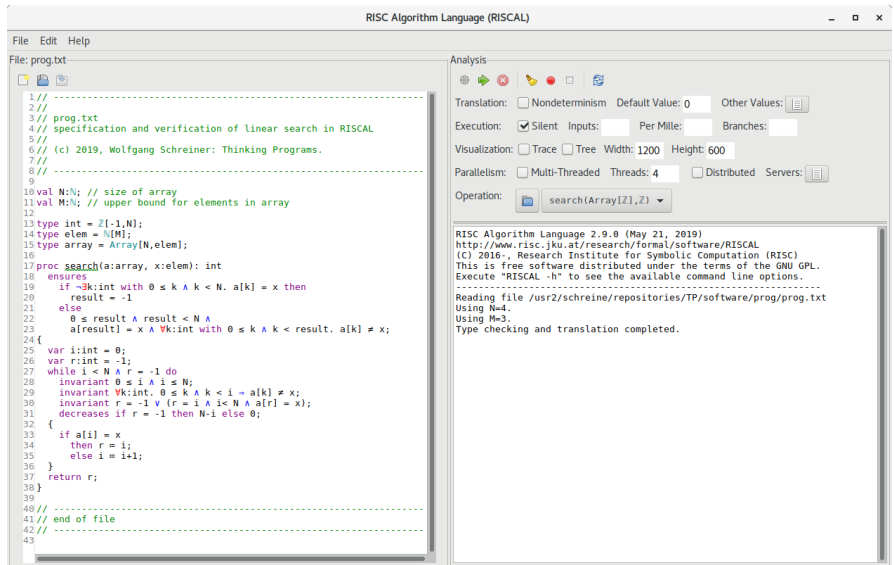


Fig. 8.11 The RISCAL Graphical User Interface

## RISCAL

RISCAL (RISC Algorithm Language) is a software system with a graphical user interface; when started from the command line with the name `prog.txt` of a text file as an argument, the system opens the window depicted in Figure 8.11. This window displays the contents of the file in an editor area to the left and various control elements and an output area to the right.

The file contains the model of a variant of the linear search algorithm that was specified in Section 8.1 and verified in Section 8.5. This model starts with the declaration of various constants and types that describe the domain in which the algorithm operates:

```

val N:ℕ; val M:ℕ;
type int = ℤ[-1,N];
type elem = ℕ[M];
type array = Array[N,elem];

```

Here the constant  $N$  denotes the length of the arrays we consider; their elements are natural numbers up to the maximum value  $M$ . Furthermore *int*, *elem* and *array* denote the types of array indices (including  $-1$  and  $N$ ), of array elements, and of the arrays themselves, respectively. The specification may contain Unicode symbols such as  $\mathbb{N}$  and  $\mathbb{Z}$ ; these symbols may be generated by typing in the editor `Nat` respectively `Int` and then pressing the key shortcut `<Ctrl>+#`.



The remainder of the file contains the declaration of a procedure annotated with its contract:

```

proc search(a:array, x:elem): int
  ensures
    if  $\neg\exists k:\text{int with } 0 \leq k \wedge k < N. a[k] = x$  then
      result = -1
    else
       $0 \leq \text{result} \wedge \text{result} < N \wedge$ 
       $a[\text{result}] = x \wedge \forall k:\text{int with } 0 \leq k \wedge k < \text{result}. a[k] \neq x;$ 
{ ... }

```



This contract specifies no precondition (which is equivalent to the precondition “true”) but by the clause *ensures* a postcondition; in the postcondition, the special name *result* refers to the return value of the procedure. Thus, if the given argument array *a* does not contain at any position the element *x*, the procedure returns the value  $-1$ ; otherwise it returns the smallest index at which *x* occurs in *a*.

The body of the procedure marked as “...” is given below:

```

var i:int = 0;
var r:int = -1;
while i < N  $\wedge$  r = -1 do
{
  if a[i] = x
    then r := i;
    else i := i+1;
}
return r;

```

This body represents essentially the command given in Section 8.5. To check the correctness of the procedure, we open via the “Other Values” button  a menu that allows us to give values to the model constants, e.g.,  $N := 4$  and  $M := 3$ . We then select in the “Operation” panel the operation *search* and press the “Start Execution” button . This lets the procedure run for *all* possible values of its input parameters, which produces the following output:

```

Executing search(Array[Z],Z) with all 1024 inputs.
Execution completed for ALL inputs (70 ms, 1024 checked, 0 inadmissible).

```

In the course of these executions the postcondition was checked for every result value of the procedure. For instance, if we change the test  $a[i] = x$  erroneously to  $a[i] \neq x$ , the checker produces the following error message:

```

ERROR in execution of search([0,0,0,0],0): evaluation of
  ensures if  $\neg(\exists k:\text{int with } (0 \leq k) \wedge (k < N). (a[k] = x))$  then ...
at line 18 in file prog.txt:
  postcondition is violated by result -1 for application search(...)

```

To further validate the correctness of the procedure, we annotate its core loop with an invariant and a termination measure:

```

while i < N ∧ r = -1 do
  invariant 0 ≤ i ∧ i ≤ N;
  invariant ∀k:int. 0 ≤ k ∧ k < i ⇒ a[k] ≠ x;
  invariant r = -1 ∨ (r = i ∧ i < N ∧ a[r] = x);
  decreases if r = -1 then N-i else 0;
{ ... }

```

Here the invariant is represented by the conjunction of the formulas specified in the multiple invariant clauses; the termination measure is specified by the term in the decreases clause. With these annotations, every execution of the procedure now also checks whether the invariant holds before and after every loop iteration and whether every loop iteration decreases the termination measure but does not make it negative. Thus, if we introduce the same error as above, we get now the following message:

```

ERROR in execution of search([0,0,0,0],0): evaluation of
  invariant ∀k:int. ((0 ≤ k) ∧ (k < i)) ⇒ (a[k] ≠ x);
at line 29 in file prog.txt:
  invariant is violated

```


Likewise, if we introduce by a clause decreases N-i a wrong termination measure, we get the following message:

```

ERROR in execution of search([0,0,0,0],0): evaluation of
  decreases N-i;
at line 31 in file prog.txt:
  variant value 4 is not less than old value 4

```

By such checks we may validate the loop annotations, in particular we may ensure that an invariant is not too strong, i.e., that it is not violated by any iteration of the loop. However, this does not yet rule out that the invariant is too weak, i.e., that it would not be able to carry a proof-based verification.

To further increase our confidence in the adequacy of the program and its specification, we press the “Show/Hide Tasks” button , which opens an additional panel depicted in the left part of Figure 8.12. This panel lists a number of tasks that are still “open” (the red color indicates that the tasks have not yet been performed). Apart from “Execute Operation” (which we have already demonstrated above), there are essentially two sets of tasks, those for validating a specification, and those for verifying the procedure.

The tasks listed under “Validate Specification” allow us to investigate the adequacy of the procedure contract (i.e., to ensure that it really expresses our intentions). For instance, the task “Execute Specification” generates from the contract the implicitly specified function

```

fun _search_0_Spec(a:array, x:elem): int = choose result:int with
  if ... then result = (-1) else ... ;

```

If we execute this operation by a double-click (with the options “Nondeterminism” selected and “Silent” not selected), the system prints out all pairs of input/output values admitted by the specification:



**Fig. 8.12** Performing the Validation and Verification Tasks

```

Executing search(Array[Z],Z) with all 1024 inputs.
Run 0 of deterministic function search([0,0,0,0],0):
Result (0 ms): 0
Run 1 of deterministic function search([1,0,0,0],0):
Result (0 ms): 1
...
Run 5 of deterministic function search([1,1,0,0],0):
Result (0 ms): 2
...
Run 1023 of deterministic function search([3,3,3,3],3):
Result (0 ms): 0
Execution completed for ALL inputs (6757 ms, 1024 checked, ...).

```

Likewise, the task “Is result uniquely determined?” generates a theorem

```

theorem _search_0_PostUnique(a:array, x:elem) ⇔
  ∀result:int with ... . (∀_result:int with ... . (result = _result));

```

whose validity we may check to ensure that the specification does not allow for one input multiple outputs:

```

Executing _search_0_PostUnique(Array[Z],Z) with all 1024 inputs.
Execution completed for ALL inputs (82 ms, 1024 checked, 0 inadmissible).

```

The majority of the tasks, however, deals with the validity of the verification conditions generated from the program and its annotations. For this

purpose, RISCAL implements a variant of the weakest precondition calculus introduced in Section 8.3; this variant does not result in a single monolithic verification condition but in a lot of small conditions; if a particular condition is not valid, the source of the error thus becomes easier to find. The condition “Is result correct?” is the core condition: it checks whether the precondition of the procedure implies the weakest precondition calculated from the body of the procedure. The tasks listed under “Verify implementation precondition” are side conditions that check whether all operations in the procedure are well-defined, i.e., whether they are only applied to legal arguments.

The conditions listed under “Verify iteration and recursion” deal with the verification of loops and recursive procedures: in our example, they check whether the various invariants hold in the initial state of the loop and whether every branch of the loop body preserves their validity and decreases the termination measure. By single-clicking a task the relevant part of the code is high-lighted, by double-clicking its validity is checked. For example, if we single-click the first task labeled “Is loop invariant preserved?”, the following piece of code is highlighted:

```
var i:int = 0;
var r:int = -1;
while i < N ∧ r = -1 do
  invariant 0 ≤ i ∧ i ≤ N;
  invariant ∀k:int. 0 ≤ k ∧ k < i ⇒ a[k] ≠ x;
  invariant r = -1 ∨ (r = i ∧ i < N ∧ a[r] = x);
  decreases if r = -1 then N-i else 0;
{
  if a[i] = x
  then i = i;
  else i = i+1;
}
return r;
```

The verification task thus checks the preservation of the first invariant in the first branch of the loop body. If we double-click the task, the validity of the condition is checked:

```
Executing _search_0_LoopOp4(Array[Z],Z) with all 1024 inputs.
Execution completed for ALL inputs (164 ms, 1024 checked, ...).
```

If we select by a right-click the top-level task folder a menu pops up with an entry “Execute all tasks”. By selecting this entry, all tasks are closed in a few seconds and thus turn blue as indicated in the right part of Figure 8.12.

By the fully automatic checking of some small models, RISCAL allows to quickly *falsify* programs, in particular also to find errors and inadequacies in their contracts and their formal annotations. This is much easier than finding the source of errors from failed proof attempts, which may be due to errors in the models, but (more often than not) also due to inadequacies in the proof strategy respectively proof automation. However, to ultimately *verify* a program, i.e., to show that it is correct in all models of arbitrary size, indeed proof-based verification is required.