A Survey Of Verification Tools Based On Hoare Logic

Nahid A. Ali

College of Computer Science & Information Technology, Sudan University of Science & Technology, Khartoum, Sudan

ABSTRACT

The quality and the correctness of software has a great concern in computer systems. Formal verification tools can used to provide a confidence that a software design is free from certain errors. This paper surveys tools that accomplish automatic software verification to detect programming errors or prove their absence. The two tools considered are tools that based on Hoare logic namely, the KeY-Hoare and Hoare Advanced Homework Assistant (HAHA). A detailed example on these tools is provided, underlining their differences when applied to practical problems.

KEYWORDS

Hoare Logic, Software Verification, Formal Verification Tools, KeY-Hoare Tool, Hoare Advanced Homework Assistant Tool

1. Introduction

Computer systems correctness is vital in the modern information society. Despite the fact that modern computer systems are composed of complex hardware and software components, verifying the correctness of the software part is often a greater problem than that of the hardware. It is known that manual inspection of complex software is prone to error and expensive, therefore tool support is required. There are many tools that attempt to discover design errors using test vectors by examining certain executions of a software system. On the other hand, formal verification tools are used to check the behaviour of a software design for all input vectors. A large number of formal tools for checking functional design errors in hardware are available and widely used. In contrast, markets of tools that check software quality are still in a beginning stage. Software quality is currently being subjected to broad research [1].

According to practical experience, using Hoare logic for verifying the correctness of computer programs is tedious, boring, obscure and entirely impractical process. This is because Hoare logic principles are generally not well-understood. In addition the whole concept is believed by a number of scholars as only a strange part of computer science history despite its fundamental role in the constantly developing field of program verification.

No doubt that this worthless state of affairs has numerous reasons. However there is one reason or issue that has greatly led to this problem: verification tools used in applying Hoare logic. These tools, in many cases, consist of a pen and few sheets of paper, making it a tedious task to verify a whole program using a pen and a sheet of paper. It is more difficult than checking the correctness of the very same program in a less formal way. Even worse, both approaches are argued to have similar chances of making a mistake. Performing the whole logical inference on paper, without using the computer (save, perhaps, in the matter of typesetting) supports the view that Hoare logic is an impractical verification tool. A direct way to solve this problem is to use automated formal verification software to facilitate checking the correctness of Hoare programs [1].

In this paper we survey the tools that accomplish automatic software verification to detect programming errors or prove their absence based on Hoare logic. The two tools we consider here are: the KeY-Hoare and HAHA tool. We begin with detailed description of Hoare logic in (Section 2). In the second part of the survey we address the KeY-Hoare tool (Section 3.1). We dedicate the third part of the survey to HAHA tool (Section 3.2) and the last part to discuss the evaluation of the two tools (Section 4).

2. HOARE LOGIC

Hoare logic (Floyd–Hoare logic or Hoare rules) is a formal system with a set of logical rules for reasoning about the correctness of computer programs. It was proposed by the British computer scientist and logician C. A. R. Hoare in 1969, and subsequently refined by Hoare and other researchers [2]. However, the original ideas were seeded by the work of Robert Floyd, who had published a similar system for flowcharts [3].

2.1. Hoare Triple

The essential feature of Hoare logic is the Hoare triple, which describes how the execution of a piece of code changes the state of the computation. A Hoare triple takes the following form:

$$\{p\}\ S\ \{q\} \tag{1}$$

Where S is a programming language statement or command, p and q are *assertions* (variables of) on the space of the program. Such Formulas are called Hoare formulas. p Indicates the *precondition* and q indicates the *postcondition*: when the precondition is met, the command establishes the postcondition. Assertions are formulas in predicate logic. Standard Hoare Logic proves only partial correctness, while termination needs to prove separately. Thus the intuitive reading of a Hoare triple is:

Whenever p *holds* of the state before the execution of S, then q will hold afterwards, or S does not terminate. Note that if S does not terminate, then there is no "after", so q can be any statement at all. Actually, one can choose q to be false to express that S does not terminate. Total correctness can be also proven with an extended version of the while rule.

It is important to note that, Hoare logic provides axioms and inference rules for all the constructs of a simple imperative programming language.

2.2. An Inference System

An inference system is a system for inferring conclusions from hypotheses in a systematic manner. Such a system can be defined by means of the following artifacts:

- A set *F* of (syntactically defined) formulas.
- A subset *A* of *F*, called the set of axioms, which includes formulas that one assumes to be valid by hypothesis.
- A set of inference rules, denoted by *R*, where each rule consists of a set of formulas called the premises of the rule, and a formula called the conclusion of the rule. One interprets a rule to mean that whenever the premises of a rule are valid, so is its conclusion. Usually, the rule is represented by listing its premises above a line and its conclusion below the line.

- An inference in an inference system is an ordered sequence of formulas, say $v_1, v_2, \ldots v_n$. Each formula in the sequence, say v_i, is either an axiom or the conclusion of a rule whose premises appear prior to v_i, i.e. amongst $v_1, v_2, \ldots v_{i-1}$. A theorem of a deductive system is any formula that appears in an inference.

This section introduces an inference system that is used to establish the validity of Hoare formulas by induction on the complexity of the program component of the formulas [4]. To this effect, one presents in turn, the formulas, then the axioms, and finally the rules.

- **Formulas**. Formulas of the inference system include all the formulas of logic, as well as Hoare formulas.

- **Axioms**. Axioms of the inference system include all the tautologies of logic, as well as the following formulas:
    - $\{false\}\ S\ \{q\}$, for any program S and any postcondition q.
    - $\{p\}\ S\ \{true\}$, for any program S and any precondition p.

- **Rules**. One presents below a rule for each statement of a simple C-like programming language, in addition to a consequence rule that allows him/her to generalize a pre/post specification.

    1. Assignment Statement Rule: One considers an assignment statement that affects a program variable (and implicitly preserving all other variables), and interprets it as an assignment to the whole program state (changing the selected variable and preserving the other variables), which is denoted by $S = E(S)$, where S is the state of the program. One has the following rule,

    $$\frac{p \Rightarrow q(E(S))}{\{p\}\ S = E(S)\ \{q\}} \qquad (2)$$

       Interpretation: If one wants q to hold after execution of the assignment statement, when S is replaced by $E(S)$, then $q(E(S))$ must hold before execution of the assignment; hence the precondition p must imply $q(E(S))$.

    2. Sequence Rule: Let S be a sequence of two subprograms, say $S1$ and $S2$. One has the following rule,

    $$\frac{\begin{array}{c}\exists\ int: \\ \{p\}\ S1\ \{int\} \\ \{int\}\ S2\ \{q\}\end{array}}{\{p\}\ S1; S2\ \{q\}} \qquad (3)$$

       Interpretation: If there exits an intermediate predicate int that serves as a postcondition to $S1$ and a precondition to $S2$, then the conclusion is established.

    3. Conditional Rule: Let S be a conditional statement, of the form: **if** (condition) **then** statement. One has the following rule,

$$\frac{\begin{array}{c}\{p \wedge t\}\; S\; \{q\} \\ (p \wedge \neg t) \Rightarrow q\end{array}}{\{p\}\; if\; (t)\; then\; S\; \{q\}} \tag{4}$$

4. <u>Alternation Rule:</u> Let S be an alternation statement, of the form: **if** (condition) **then** statement **else** statement. One has the following rule,

$$\frac{\begin{array}{c}\{p \wedge t\}\; S1\; \{q\} \\ \{p \wedge \neg t\}\; S2\; \{q\}\end{array}}{\{p\}\; if\; (t)\; then\; S1\; else\; S2\; \{q\}} \tag{5}$$

5. <u>Iteration Rule</u>: Let S be an iterative statement, of the form: **while** (condition) statement. One has the following rule,

$$\frac{\begin{array}{c}\exists\; inv: \\ p \Rightarrow inv \\ \{inv \wedge t\}\; S\; \{inv\} \\ inv \wedge \neg t \;\Rightarrow q\end{array}}{\{p\}\; while\; (t)\; S\; \{q\}} \tag{6}$$

Interpretation: The first and second premises establish an inductive proof to the effect that predicate inv holds after any number of iterations. The third premise provides that upon termination of the loop, the combination of predicate inv and the negation of the loop condition must logically imply the postcondition. Predicate inv is called an invariant assertion. It must be chosen in order to be sufficiently weak to satisfy the first premise, yet sufficient strong to satisfy the third premise (and the second).

6. <u>Consequence Rule</u>: Given a Hoare formula, one can always strengthen the precondition and/or weaken the postcondition. One has the following rule:

$$\frac{\begin{array}{c}p \Rightarrow p' \\ q' \Rightarrow q \\ \{p'\}\; S\; \{q'\}\end{array}}{\{p\}\; S\; \{q\}} \tag{7}$$

Using the proposed axioms and rules, one can generate theorems of the form $\{p\}\; S\; \{q\}$.

3. Tools For Formal Program Verification

Generally, Formal program verification tools are classified into three categories: interactive or semi-automatic proof construction environments such as Isabelle [5] and the Prototype Verification System [6], tools for model checking (using abstract interpretation and similar techniques) such as the Symbolic Model Verifier [7], and systems based on the Hoare calculus or related approaches (weakest precondition, dynamic logic) such as the Frege Program Prover [8], Perfect Developer [9], the KeY-Hoare [10] and Hoare Advanced Homework Assistant (HAHA) [11].

Since we concentrate on structured program development using invariants, pre and post conditions we selected the latter two systems for survey. Beneath, we give detailed descriptions of the selected two tools.

3.1. The Key-Hoare Tool

KeY-Hoare [10] is built on top of the software verification tool KeY (which is considered one of the most powerful verification systems for Java developed jointly by groups at Universit¨ at Karlsruhe (Germany), Universit¨ at Koblenz-Landau (Germany), and Chalmers University of Technology (Sweden). It is distributed under the Gnu General Public License and features a Hoare calculus with state updates. The KeY-Hoare tool is available for free and can be installed easily. It is a verification system that utilizes a variant of Hoare logic with explicit state updates which enable users to reason about a program correctness by means of symbolic forward execution. Differently, the assignment rule in more traditional Hoare logics requires less natural and harder to learn backwards reasoning. No explicit weakening rule is required and first-order reasoning is automated. The tool is appropriate for teaching program verification, because students can concentrate on reasoning about programs, after their natural control flow and proofs are checked by the computer [10]. At the present time, the GUI of the KeY-Hoare tool contains several elements inherited from the full Java version. These elements are not useful in more specialized contexts which should be firstly cleaned up and simplified. The current version of KeY-Hoare does not support arrays since Java arrays are too complicated [10]. It provides many features including partial correctness proofs, total and execution time correctness proofs and integer and Boolean typed arrays. It is worthy to note that the logic behind the tool is not pure Hoare logic, but Hoare logic with updates. Therefore one difficulty, in the adoption of the tool, is that it is necessary to extend the Hoare logic formula to include the updates. Another difficulty with adoption of KeY-Hoare tool is that it exposes users to the KeY prove where numerous logic rules are applied at each step of verification. Users must be instructed to decide which of these logic rules should be applied and which of them is to be avoided in typical cases [12].

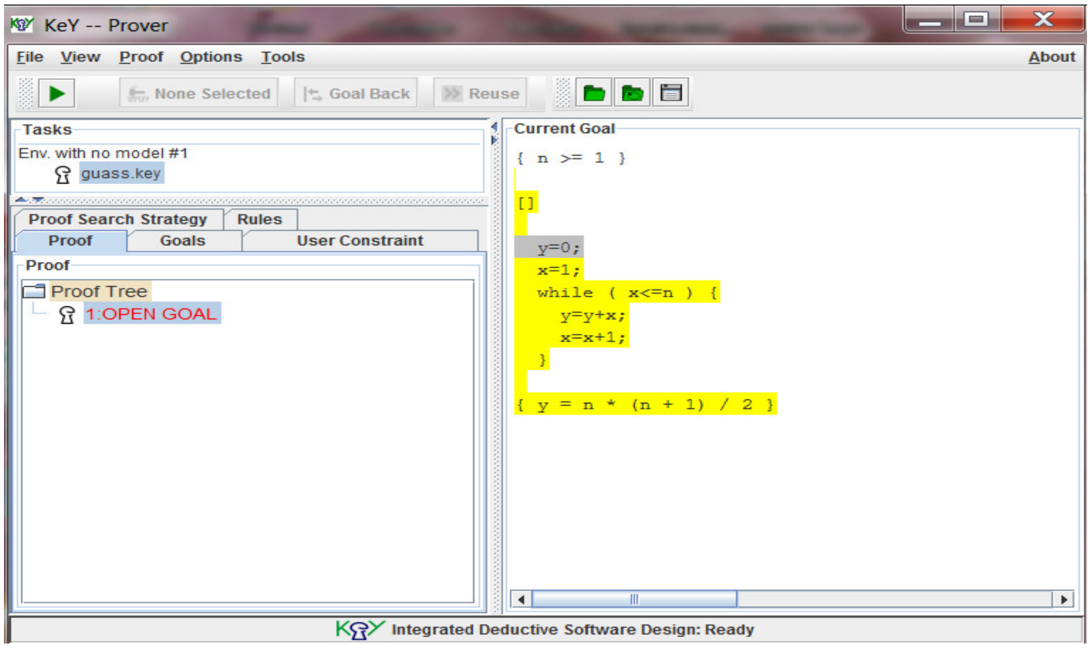
3.1.1 Using Key-Hoare Tool

KeY-Hoare is available from a website indicated in [13]. Besides compilation from the source code, providers offer a pre-compiled byte-code version and installation via Java Web start technology. Detailed installation instructions are also supplied by the website.

Input files for KeY-Hoare must have either **.key** or **.proof** as file extension. Conventionally, **.key** files contain only the problem specification, i.e., the program together with its specification. In contrast **.proof** files include proofs (or proof attempts) and are created when saving a proof.
An example that illustrates the format is shown in Figure 1. An input file consists of three sections, these are:

1) A section starting with keyword \functions declares all required rigid function symbols used.
2) A section starting with keyword \programVariables declares all program variables used in the program. Local variables declarations within the program are not allowed. However, multiple declarations are permitted.
3) A section starting with keyword \hoare contains the Hoare triple with updates to be proven valid, i.e., it contains the program and its specification. The initial update usually contains an assignment of fixed but arbitrary logical values to the input variables of the program. As illustrated in Figure 3.1, we do not provide the update rule because we need the program to be similar to original Hoare logic.

After starting KeY-Hoare, the prove window becomes visible (the screenshot is displayed in Figure 1). The prove window consists of a menu and toolbar, a status line and a central part split into a left and a right pane. The upper left pane displays a list of all loaded problems. The lower left pane offers different tabs for proof navigation or strategy settings. The right pane displays the currently selected sub-goal or an inner proof node. Before we explain the various sub-panes in more detail, our first task is to load a problem file. We can do this either by selecting Load in the File menu or by clicking on the icon that browse and load problem or proof files in the toolbar.



```
{ n >= 1 }

[]

  y=0;
  x=1;
  while ( x<=n ) {
    y=y+x;
    x=x+1;
  }

{ y = n * (n + 1) / 2 }
```

Figure 1: Screen Shot of KeY- Hoare System

After loading the file, the right pane of the prove window displays the Hoare triple as specified in the input file. The proof tab in the left pane should display the proof tree consisting of a single node. The first time during a KeY-Hoare session, when a problem file is loaded, the system loads a number of libraries. This loading takes a few seconds.

The upper part of the left pane displays all loaded problems while the lower part provides some useful tabs, these are [12]:

- *The Proof tab* that shows the constructed proof tree. A left click on a node updates the right pane with the node's content (a Hoare triple with updates). Using a right click offers a number of actions like pruning, searching, etc.
- *The Goals tab* that lists all open goals, i.e., the leaves of the proof tree that remain to be justified.
- *The Proof Search Strategy tab* that allows tuning automated proof search. The strategy for KeY-Hoare only allows adjusting the maximal number of rule applications before an interactive step is required, and (de-)activation of the auto-resume mode.

Excluding the above tabs, all other tabs are of no importance for KeY-Hoare. Therefore, they will be removed in future versions (as stated by the developer of KeY- Hoare).
The right pane displays the content of a proof node in two different modes. This depends on whether the node is (a) an inner node or a leave justified by an axiom or (b) it represents an open proof goal.

(a) The mode of "Inner Node View" is used for inner nodes of the proof tree. It highlights the formula which had been in focus at time of rule application as well as possible necessary side formulas. The applied rule is listed on the bottom of the view.

(b) The mode of “Goal View” is used when an open goal is selected. This view shows the Hoare triple to be proven and allows users to apply rules. Moving the mouse cursor over the expressions within the node highlights the smallest enclosing term or formula at the current position. A left click creates a popup window showing all applicable rules for the currently highlighted formula or term.

We illustrate how the system is used by proving correctness of a small program given in a number *n* returns the sum of all subsequent numbers from *1* to *n* inclusively. The code will look like this:

```
\functions {
}
\programVariables {
  int x;
  int y;
  int n;
}
\hoare{
{n >= 1}
\[{
          y = 0;
          x = 1;
          while (x <= n)
          {
                    y = y + x;
                    x = x + 1;
          }
}\]
{
  y= n * (n+1) / 2
}
}
```

Figure 2: KeY-Hoare Input File for the Gauss Example.

All variables are integers. Provided that the starting value of *n* is nonnegative, the program always terminates with the value of *y* consistent with gauss formula (y= n * (n+1) / 2). A suitable precondition is *n* >= 1. The postcondition can be stated simply as y= n * (n+1)/2. The initial Hoare triple with updates reads as follows:

{n >= 1} [ ] y = 0;   x = 1; while (x <= n) {y = y + x; x = x + 1;} { y= n * (n+1) / 2}
A file with an initial Hoare triple as proof obligation should be first loaded to the KeY-Hoare system. Then users can select a rule offered in a popup-menu after moving the mouse pointer over a Hoare triple and clicking. There is exactly one applicable rule for each program construct and the system offers typically this rule: users experience statement-wise symbolic execution of the target program. The only non-trivial interaction is to supply an invariant in a dialogue box that opens when the loop rule is applied. The invariant y = x* (x-1) / 2 & x<= n+1 & n >= 1 is sufficient.

Whenever reaching first-order verification conditions, the system offers a rule Update Simplification that applies the update rules automatically. At this point, users can opt to push the green Go button. Then the built-in first-order theorem prover tries to establish validity automatically. If no proof is found, typically, the invariant or the specification (or the code!) is too weak or simply wrong. Inspecting the open goals usually gives users a good hint. The system allows users to follow symbolic execution of the program and to concentrate on getting invariants and specification right. First-order reasoning is thus left to the system. It is possible for users to inspect and undo previous proof steps as well as to save and load proofs.

Regarding the automation, up to this point the required interactive steps consisted of manual application of program rules and invocations of the strategies to simplify/ prove pure first-order problems. In order to avoid starting the strategies manually users can activate the auto-resume mode. This invokes the strategies on all open goals after each manual rule application and simplifies them as far as possible. In standard mode they will not apply program rules. While performing a proof it is possible to save the current state at any time and to load it afterwards. For this users have to select Save in the File menu and enter a file name ending with **.proof**.

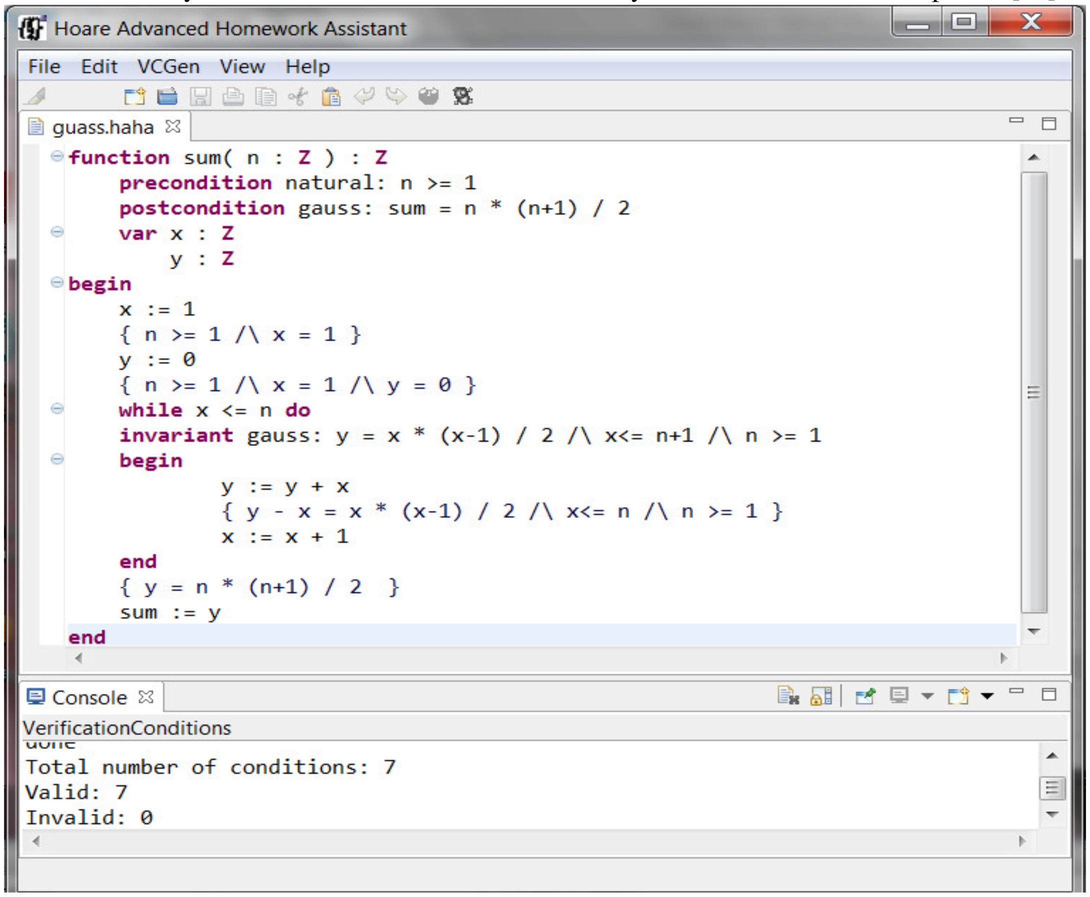
3.2. Hoare Advanced Homework Assistant (HAHA) TOOL

HAHA is a programming language firmly fixed in the new program development environment based on Eclipse [12]. Its purpose is to allow students to learn Hoare logic. A user can write simple programs and annotate them with Hoare logic assertions. Then the HAHA environment verifies the assertions against the code and discharges them with aid of external theorem provers, both automatically and interactively. Users can write programs that operate on true integers and on arrays. HAHA supports proofs of both partial and total correctness. HAHA relies on a SMT solver called is Microsoft Z3 [14] to prove the validity of generated formulae.

In the user interface of HAHA, shown in Figure 3, the main pane of the window is filled with the source code of the program that users work with. The interface shows an editor for simple while programs. It has all features expected from a modern IDE including syntax highlighting, automated completion proposals and error markers. Once a user enters a program, it is processed by a verification condition generator that implements the rules of Hoare logic. The resulting formulae are then passed to an automated prover. If the solver is unable to specify the correctness of the program, error markers are generated to direct users to the assertions which could not be proven. A very useful feature of HAHA is its ability to find counterexamples for incorrect assertions. These are included in error descriptions displayed by the editor. The input language of HAHA is that of while programs over integers and arrays. It's designed so that its mechanisms and data types match those supported by state of the art satisfiability solvers, e.g. Z3 [14] or CVC4 [15].

A notable strength of HAHA is that the syntax of the programs is close to original Hoare logic with possible extensions, but in a manner that is easily digestible by users who are familiar with programming languages such as Pascal or Java. Added to that, the process of verification gives the impression that it is carried out as part of program development, in particular the environment does not change to the one of interactive prover in order to assist in discharging verification conditions. Instead, users give assertions between instructions that are subsequently verified by an automatic theorem prover [12]. Another strength of HAHA is that it is relatively easy adaptable to any teaching environments.

In contrast to the common trend to make only loop invariants obligatory, HAHA compels students to fill in all intermediate assertions. This might seem surprising, because the requirement to write many formulae increases the amount of work necessary to create a verified program. However, in the case of a teaching aid, this approach is more beneficial. First, this suggests students to match the assertions with relevant Hoare logic rules. In this way it reinforces the process of teaching the logic. Second, it gives the students a tangible experience of how much information must be maintained at each step of the program in order to execute it correctly— the process of making the verification work also gives students a tangible experience, making them see how it is easy to overlook transformation of some tiny detail in this information packet [12].



```
function sum( n : Z ) : Z
    precondition natural: n >= 1
    postcondition gauss: sum = n * (n+1) / 2
    var x : Z
        y : Z
begin
    x := 1
    { n >= 1 /\ x = 1 }
    y := 0
    { n >= 1 /\ x = 1 /\ y = 0 }
    while x <= n do
    invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
    begin
            y := y + x
            { y - x = x * (x-1) / 2 /\ x<= n /\ n >= 1 }
            x := x + 1
    end
    { y = n * (n+1) / 2  }
    sum := y
end
```

```
Total number of conditions: 7
Valid: 7
Invalid: 0
```

Figure 3: Screen Shot of HAHA System

3.2.1 Using HAHA TOOL

To run HAHA, launch the haha executable from the installation directory. HAHA is typical file editor with standard and intuitive commands. It is important to note that the source files should have the extension **.haha** in order for the editor to function properly. When using the correct file extension, the code is highlighted. To start the verification process, a user can select the "**Generate VCs"** command from the main menu or the toolbar. This command can be also accessed by right-clicks anywhere in the editor and chooses "**Generate VCs"** from the displayed menu. HAHA will then show a console with computed verification conditions and messages logged during verification. An error markers will be added to the editor if there were any problems.

Assume that a user wants to write a function that given a number *n* returns the sum of all subsequent numbers from *1* to *n* inclusively. The code will look like this (this code is taken from Sznuk and Schubert work) [12]:

function sum(n : **Z**) : **Z**
precondition natural: n >= 1
postcondition gauss: sum = n * (n+1) / 2

```
var x : Z
y : Z
begin
x := 1
{ n >= 1 /\ x = 1 }
y := 0
{ n >= 1 /\ x = 1 /\ y = 0 }
while x <= n do
invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
begin
y := y + x
{ y - x = x * (x-1) / 2 /\ x<= n /\ n >= 1 }
x := x + 1
end
{ y = n * (n+1) / 2 }
sum := y
end
```

Figure 4: HAHA Input File for the Gauss Example.

The code should not be surprising especially for users who are familiar with Pascal as much of the syntax is based upon the language.

The code start with a header of the function: **function** sum (n: **Z**): **Z** contains the keyword *function* that informs the system to interpret the following expressions as a function. Then the name of the function is given, in this example it is *sum*. The function parameters are enclosed in the parentheses. In our example, there is one parameter that is called *n*. The type of the parameter is declared after the colon. It is Z in this case, i.e. the type of integer numbers as users know them from mathematics (these are not 32-bit integer numbers frequently met in programming languages). At the end of function header, the parenthesis with parameters is followed by the declaration of the result type. In our case again this is Z. This function header can be followed by the definition of the body.

All variables that are used in code must be declared first between the function header and the body using the keyword *var*, as in Pascal, to point to the beginning of variable declaration sequence. In our case, the declarations of the variables *x* and *y* are added. Unlike Pascal, the elements of the sequence are separated with newlines, but not with semicolons. This holds both for variable declarations and instructions. The variable assignment is similar to Pascal as in *x*: = *1*. Also the Pascal style is used to define the return result of the function, i.e.: sum: = *y*. The while loop is defined by an expression of the form:

while x <= n **do** followed by an instruction the loop iterates over. In our example this is a block of two instructions enclosed between begin and end, i.e.:

```
begin
y := y + x
x := x + 1
end
```

To express our intent with regard of the function, we can add precondition and postcondition formulas between the function header and its body. The precondition is defined with the keyword precondition, in our case it says that the function can be called when the parameter *n* is not less than 1. The postcondition, defined with the keyword postcondition, says that the result is equal to the commonly known Gauss formula i.e. closed formula for the sum of the integers from 1 to n. User can assigns names to these conditions to make future reference easier:

precondition natural: n >= 1
postcondition gauss: sum = n * (n+1) / 2

In Hoare logic each instruction must be enclosed between two assertions. The first one defines the condition of the program state expected before the instruction and the second one defines the state resulting from the execution of the instruction. The precondition-postcondition pair is the assertions for the whole function. The assertions for other instructions are placed between instructions in curly brackets. In this case, we do not write the first and last assertions in function since these are stated with precondition and postcondition respectively. Therefore, the initial assignments after the addition of the assertions look as follows:

```
begin
x := 1
{ n >= 1 /\ x = 1 }
y := 0
{ n >= 1 /\ x = 1 /\ y = 0 }
while x <= n do
```

The formula that at the entry point to the loop at each its iteration is called the loop invariant condition and it's defined with the keyword *invariant*. So, the header of while loop after adding the invariant is as follows:

```
while x <= n do
invariant gauss: y = x * (x-1) / 2 /\ x<= n+1 /\ n >= 1
```

The invariant formula can be named to make future reference more precise. Again, there is no need to mention assertions at the beginning and at the end of the loop body as they equate to the invariant.

4. EVALUATION

We evaluated the tools with respect to the following criteria [16].

1. ***Ease of use:*** Users should not be forced to spend too much time on learning a new language and on understanding its characteristics. Furthermore, the system should be able to discharge the proof obligations as automatically as possible.
2. ***Feedback on errors:*** Error messages and reports on proof failures should provide readily understood feedback without deep knowledge of the tool specific underpinning formalism. In this way users keep the focus on the verification problem itself.
3. ***Adequate documentation:*** There should be sufficient documentation that is easily understood by users. The better the documentation the less help is needed on basic language issues.
4. ***Ease of installation:*** The system should be easy to install on any of the platforms users typically use, i.e., on Windows, Linux, and Mac OS. In addition to easy installation, further software components from other sources should not be needed in installation.

In the remaining section we describe our experiences with each of the two tools.

1. **The KeY-Hoare Tool:** The evaluation was performed with version 0.1.9 of KeY-Hoare. Installation under Windows, Linux and MacOS is straightforward. The first thing we notes about KeY-Hoare is its simplicity: given a program, precondition and postcondition, the

system checks whether the program is correct with respect to its specification. However, the proofs are not fully automatic, and human intervention is needed as users must select the appropriate rule at each statement. Another reason for simplicity is that users who are acquainted with C++ will learn KeY-Hoare in a short time; the web site of KeY-Hoare offers sufficient documentation.

The work with KeY-Hoare exhibited some practical problems. KeY-Hoare assumes that the program is syntactically correct and well-typed or the program will not be loaded to the system. Another problem is that the logic behind the KeY-Hoare tool is not pure Hoare logic, but Hoare logic with updates. Therefore, the first obstacle in the adoption of the tool was the need to add the updates formula to the original Hoare triple.

2. **The HAHA Tool:** The evaluation was performed with version 0.5 of HAHA that is tested with Z3 version 4.3.0. HAHA relies on a SMT solver to prove the validity of generated formulae. Currently the only supported solver is Microsoft Z3. Installation is available under many platforms such as Windows, Linux and MacOS. Sufficient documentations are available in HAHA web site and a great support is offered by the developer of HAHA when one requests help.

   HAHA supports proofs of partial and total correctness both automated and interactive. Users can write programs that manipulate on true integers and on arrays. Also, HAHA allows users to define Boolean variables but the verification condition generation is not supported yet for Boolean type. HAHA has the ability to find counterexamples for incorrect assertions when the solver is unable to ascertain the correctness of the program.

   One of the strengths of HAHA is that the syntax of the programs is close to original Hoare logic. However, HAHA forces users to fill in all intermediate assertions between instructions. This step increases the amount of work necessary to create a verified program, since the users need to write many formulae. Another notable strength of HAHA is that the tool is easily digestible by users who are familiar with programming languages such as Pascal or Java.

For Hoare-based verification purpose, HAHA is a better choice than KeY-Hoare tool at present. It offers a high-level language that can be learned within a short time. Also, the high degree of proof automation combined with the ability to provide feedback on failed proof attempts enable users to easily use it to prove the correctness of programs. HAHA also removes the difficulty associated with the process of manually applying Hoare logic.

5. CONCLUSIONS

In this paper, we surveyed two main tools for automatic formal verification of software based on Hoare logic. We focused on tools that provide some form of formal guarantee, and therefore, help in improving software quality. A short tutorial on these tools is delivered, underlining their differences when applied to practical problems. We evaluate the tools discussed in this survey in section 4. The evaluation provides the key features of each tool and shows that HAHA is a better choice than KeY-Hoare tool because it offers a high-level language, which can be learned within a short time. Also, the high degree of proof automation together with the ability to provide feedback on failed proof attempts make users comfortable when using it to prove the correctness of their programs. HAHA also removes the difficulty associated with the process of applying Hoare logic manually.

ACKNOWLEDGEMENTS

I truly thank my supervisor Prof. Ali Mili for his continuous advice and support. Also I appreciate the assistance offered to me by Dr. Abu Obeidh.

REFERENCES

[1] D'silva, Vijay and Kroening, Daniel and Weissenbacher, Georg, "A survey of automated techniques for formal software verification." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27(7), pp.1165-1178, 2008.

[2] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, vol. 12, no. 10, pp. 576 - 580, 1969.

[3] R. W. Floyd, "Assigning Meanings to Programs," Mathematical Aspects of Computer Science, vol. 19, no. 1, pp. 19-32, 1967.

[4] Mili , Ali ; Tchier, Fairouz ;, Software Testing: Concepts and Operations, Hoboken, New Jersey: John Wiley & Sons, 2015.

[5] "Isabelle," [Online]. Available: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[6] S. Owre, J. Rushby and N. Shankar, "PVS: A Prototype Verification System," in 11th International Conference on Automated Deduction (CADE), vol. 607, Springer-Verlag, 1992, pp. 748-752.

[7] "Symbolic Model Verifier," [Online]. Available: http://www.cs.cmu.edu/~modelcheck/smv.html.

[8] J. Winkler, "The Frege Program Prover FPP," in Internationales Wissenschaftliches Kolloquium, vol. 42, 1997, pp. 116-121.

[9] D. Crocker, "Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement," Tools Exhibition Notes at Formal Methods Europe, 2003.

[10] H¨ahnle , Reiner; Bubel, Richard, "A Hoare-Style Calculus with Explicit State Updates," Formal Methods in Computer Science Education(FORMED), pp. 49-60, 2008.

[11] "Hoare Advanced Homework Assistant (HAHA)," [Online]. Available: http://haha.mimuw.edu.pl/.

[12] T. Sznuk and A. Schubert, "Tool Support for Teaching Hoare Logic," in Software Engineering and Formal Methods, Springer, 2014, pp. 332-346.

[13] "Key- Hoare System," [Online]. Available: http://www.key-project.org/download/hoare/.

[14] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337-340.

[15] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi´c, T. King, A. Reynolds and C. Tinelli, "CVC4," in Computer Aided Verification, Springer, 2011, pp. 171-177.

[16] Feinerer, Ingo and Salzer, Gernot , A comparison of tools for teaching formal software verification, Formal Aspects of Computing, vol. 21(3), pp. 293–301, 2009.

AUTHORS

Miss. Nahid A. Ali has been a member of the Faculty of Mathematical and Computer Science at University of Gezira, Sudan since 2004, where she held different positions including the Coordinator of the Postgraduate Studies. She received her Master's and Bachelor's degrees in computer science from University of Gezira, Sudan