Crafting tool support for teaching Hoare logic*

Tadeusz Sznuk

Institute of Informatics
University of Warsaw
ul. Banacha 2
02–097 Warsaw
Poland

Abstract. Currently, software certification and verification is perceived as overly difficult and hard to understand task in program development. This image can be changed through effective instruction of prospect programmers during their studies. Teaching Hoare logic is a process that can be made significantly easier if appropriate tools are employed. Having an environment tailored to the style and content of a particular lecture is especially beneficial. We argue that current state of related technology is such that it is possible to implement a customised tool with acceptable effort. We illustrate our point by showing selected aspects of the implementation of one such application, which we call HAHA (Hoare Advanced Homework Assistant).

1 Introduction

Hoare logic plays a fundamental role in the field of program verification and formal semantics. In its basic form, it is a simple and well studied [2] framework. Its numerous variants and extensions are of great use in both theoretical research in the field of formal methods and practical implementation of various verification tools.

Unfortunately, the experience of our faculty shows that this is not how a typical student sees the subject of Hoare logic. Rather, it is perceived as tedious, boring, obscure and, above all, utterly impractical. Its principles are, in general, poorly understood, and the whole concept is believed by many to be no more than a weird part of computer science history, which is inflicted upon them by their professors solely for the reason of sentiment. Even those who manage to grasp the formal rules of Hoare logic seldom see any potential uses for it in actual software development.

It is doubtless that this sorry state of affairs has multiple reasons. Some of these lie deep in the human nature, and it is beyond both our capability and the scope of this paper to rectify them. But there is one issue, which, in our opinion, contributes greatly to the aforementioned problem, that we can hope to at least partially resolve. It is the issue of tools that are used to teach Hoare logic. In many cases, said tools consist of a pen and few sheets of paper. These

* This work was partly supported by Polish government grant N N206 493138.

respectable instruments have proven to be an indispensable educational aid over centuries. And yet, in the particular application that is our matter of interest here, they appear to show some deficiencies. Verifying that a Hoare program written on a sheet of paper is correct is a very tedious task. It is much harder than proving the correctness of the very same program in a less formal way. What is worse is that the chances of making a mistake can be argued to be similar for both approaches. It is, therefore, not appropriate to blame students for failing to understand the importance of a quality assurance method which requires a very significant amount of effort for no perceived gain. The fact that the whole exercise is performed on paper, without any aid from the computer (save, perhaps, in the matter of typesetting) further enforces the view that Hoare logic is a purely theoretical subject of little importance. Tediousness of the verification process is also an issue for lecturers, as it turns authoring and grading assignments into truly formidable tasks.

The most direct way of tackling the troubles we just described is to use automated formal verification software to facilitate checking the correctness of Hoare programs. Quite a few of such tools are in existence. In fact, the area has enjoyed a rather significant progress in development during the recent years. Systems such as ESC/Java [9,22], Frama-C [7,10] or Microsoft VCC [11] have not yet made their way into the everyday toolbox of an average developer, but have proven to be usable in the verification process of various software projects. It can be safely assumed that a tool, which is capable of assuring correctness of something as complex as memory accesses performed by a hypervisor, could also be successfully employed in verification of simple programs that are used during lectures on Hoare logic.

It is somewhat surprising that, in spite of the features offered by the aforementioned tools, static verification systems are quite rarely used in the industry [18]. Existence of usable systems is not the only condition necessary to ensure widespread adoption of the formal verification techniques. It requires also awareness of the developers that such tools exists and their consciousness of the way the methods behind the tools work. Therefore, it is important to shape developers' culture so that more refined methods of software cultivation that involve reasoning on programs are well understood and perceived as practical. One important way to achieve this is to show the methods to future programmers during their curriculum in an attractive, modern way.

The problem, however, is that the design of these powerful systems is geared towards applications in large scale software development, rather than education, and these two goals are, at least in some aspects, exclusive. One of the reasons is thar rich feature set of languages such as C or Java, as well as the need to minimize the burden that the verification system puts on a programmer during creation and evolution of code, invariably result in complexity of the underlying formal logic, which prohibits its use in an introductory course.

There is a number of tools, such as KeY-Hoare [8], that are designed specifically for the purpose of being educational aids. Unfortunately, these are not as well developed as their commercially applicable counterparts. In addition,

variants of Hoare logic which are implemented by said tools are seldom compatible with these from existing teaching material. It should also be noted that, while automated systems can prove to be very helpful by alleviating the issue of the amount of work required for verification, they might also hamper student's understanding of the principles of Hoare logic. That is because the ease of experimentation afforded by such software often allows one to solve an exercise by trial and error, with little regard for the strict formal rules that govern the correctness of a solution.

In this paper, we argue that an opportunity to improve the described situation lies in the current state of the art in the areas of domain specific language design and automated satisfiability solving. Existing frameworks and tools make the implementation of a polished, usable (although only in educational setting) software verification system a relatively simple task. Creation of tools designed to suit a particular set of lectures thus becomes feasible. It is also conceivable to include the students in the development effort. We conjecture that having a chance to implement the rules of a Hoare logic in a complete and feature-rich tool, instead of just applying these rules on paper, could greatly further a student's understanding of the principles of said logic. To illustrate our claim, we present here details of the design of a verification system which we developed. Our goal is not to provide all implementation details, but rather to show the amount and kind of effort that was necessary to create the tool.

2 HAHA overview

Motivated by the difficulties we encountered when teaching Hoare logic as a part of a course in formal verification and semantics, we decided to create an interactive tool to help us. That tool, called Hoare Advanced Homework Assistant, or HAHA, is a development environment based on Eclipse. For technical reasons we choose to distribute it as a complete application, but, in principle, it could also be used as a set of Eclipse plugins. Let us now describe the nature and capabilities of the tool, and then proceed with discussion of its implementation.

HAHA is an editor for simple while programs. It has all features that are expected of a modern IDE, such as syntax highlighting, automated completion proposals and error markers. Once a program is entered, it is processed by a verification conditions generator, which implements the rules of Hoare logic. The resulting formulae are then passed to an automated prover. If the solver is unable to ascertain the correctness of the program, error markers are generated to point the user to the assertions which could not be proven. A very useful feature is the 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. We designed it so that its features and data types match those supported by state of the art satisfiability solvers. As the language is fairly standard, we refrain from giving its actual grammar, although fragments of it can be found in

later sections. Instead, we present an example of a program, and discuss some finer features of HAHA syntax that it shows.

```
predicate sample(x : Z) = (x = 42)
axiom test: forall x: Z y : Z, (x + y)^2 = x^2 + 2*x*y + y^2

function nsum(n : Z) : Z
  precondition n_is_positive : n > 0
  postcondition gauss : nsum = n * (n - 1) / 2
var
  i : Z
  s : Z
begin
  s := 0
  { n_is_positive: n > 0 }  { s = 0 }
  i := 0
  { n_is_positive: n > 0 } { s = 0 } { i = 0 }
  while i < n do
    invariant n_is_positive: n > 0
    invariant i_le_n : i <= n
    invariant gauss: s = i * (i - 1) / 2
    counter L : Z [L - 1], L = n - i
  begin
    s := s + i
    { count: L = n - i } { n_is_positive: n > 0 }
    { i_lt_n : i < n } { s = i * (i + 1) / 2 }
    i := i + 1
  end;
  { s = n * (n - 1) / 2 }
  nsum := s
end
```

Fig. 1. Example program. The function computes the sum of numbers from 1 to n using a loop. The precondition states that the final result is same as would be obtained from Gauss' formula

The program consists of a single function, which calculates the sum of natural numbers from 1 to n, for a given value of n. The specification simply states that the result (which, just like in Pascal, is represented by a special variable) matches the well known Gauss' formula, as long as the argument is not negative. Here it must be noted that the type `Z`, used in the example program, represents unbounded (that is, arbitrarily large) integers. This is one example of a design choice that would not necessarily be valid for a verifier meant to be used in actual software development. The reason is that errors related to arithmetic overflows are quite common, and are often exploited for malicious purposes. It seems reasonable to require a static analyser to be able to ensure that no such

mistakes are present in the checked code. In our setting, these considerations play a less important role, so we were able to choose a simpler model of arithmetic. On the other hand, it might be actually desirable to be able to illustrate difficulties associated with the necessity of avoiding or handling overflows. For this reason we have created a modification of our tool to allow the use of `Int` variables, modeled as 32-bit vectors. The fact that we were able to add such feature with relative ease seems to reinforce the claims which were stated in the introduction.

Structure of the language appears to be fairly self explanatory. Let us note that, following the example of Eiffel, loop invariants can be named. This is also extended to other types of assertions. The names are useful for documentation purposes, and make error messages and solver logs much easier to read. It is possible to give multiple invariants for a single loop. Similarly, a sequence of assertions is interpreted as a conjunction. This is notably different from the approach taken in some textbooks and course material, in which an implication is supposed to hold between two consecutive assertions not separated by any statement. We believe the former interpretation to be clearer, but, to illustrate the ease with which HAHA can be modified to fit the requirements of a particular course, we have implemented the latter in a variant of our tool. Finally, let us remark that an explicit application of the weakening rule can be unambiguously represented with the help of the `skip` statement.

In the example program, each pair of consecutive statements is separated by assertions. Rules of the Hoare logic allow these midconditions to be inferred automatically, as long as loop invariants are provided. In HAHA, however, no such inference is performed. All assertions must be explicitly stated in the program code. This might seem surprising, since the need to write many, often trivial, formulae increases the amount of work necessary to create a verified program. Practical verification tools are, typically, able to infer most assertions, and sometimes even fragments of the loop invariants. In spite of this, we believe that, in the case of a teaching aid, our approach is more beneficial. The reason is that it forces students to more precisely understand what is the set of states reachable at a given program point, and what exactly must be stated about it in order to make the program correct with regard to formal rules of the Hoare logic. Finally, let us note that addition of an assertion inference mechanism is another example of a simple modification that can be made to tailor the tool to custom requirements (although we have not performed this particular experiment).

HAHA supports proofs of both partial and total correctness. To facilitate the latter, we allow an invariant to be parametrized by an integer variable. The conditions generated for such an invariant, which we call a counter, ensure that the loop condition holds precisely when the invariant formula is true for an argument of 0. It must also be proved that, if the invariant holds before the loop for an argument of $L \in \mathbb{N}$, it will, after the loop body is executed, hold for a number $L' \in (N)$ strictly smaller than L. Concrete syntax used by HAHA for the purpose of termination proving can be seen in the example program, although proving termination of this particular loop is not a very challenging task. The expression in square brackets represents the aforementioned value of

L'. As a side note, we are also working on adding support for binary assertions (and corresponding total correctness rules) to HAHA.

Before we conclude this brief examination of the capabilities of HAHA, let us turn our attention to the very first lines of the example program. They contain definitions of an axiom and a predicate. They are provided only as an illustration, and are not actually used in the rest of the code. When creating specifications for more complex procedures, however, it is often desirable to use predicates to shorten and simplify notation. Our experiences show that this is especially useful in code operating on arrays, as it tends to employ rather complex assertions. Axioms are also helpful, especially when the automated solver is unable to prove a formula that we know to be true. We shall return to this issue when discussing details of the solver technology employed by HAHA.

3 Implementation of XText-based editors

In this section, we focus on the structure and implementation of various components of the Hoare Advanced Homework Assistant. In particular, we are interested in modules responsible for parsing, scoping rules, validation and compilation, which, in our case, amounts to verification condition generation. These are the principal elements from which a typical XText based development environment is built. Our discussion is quite detailed, since it is our intention to allow the reader to estimate costs and effort necessary to create or modify a tool similar to HAHA. We focus the presentation of each component on details of its implementation that are specific to usable graphical development environments, as opposed to simpler, command line compilers.

The base tool that connects all HAHA components is XText [14]. It is an open-source Eclipse framework for development of programming languages. It was originally meant to be used for simple domain specific languages, rather than full-fledged ones, such as C or Java. While its design has evolved, and it can now be used to create quite complex systems, it is still best suited for situations when the project is small, or the language's syntax, semantic and scoping rules can be modified to make best use of the default behaviour provided by XText. HAHA appears to be a prime example of such a project. The application is pretty modest in size and required features, and details of the language design can be adjusted to make editor implementation easier. Drawbacks of the XText framework are less prominent in this kind of system, and its advanced capabilities allow us to create a modern and feature rich development environment with little effort.

Let us now proceed with description of the parsing process, which is, quite naturally, guided by a context-free grammar. The flavour of grammars used by XText is similar to that found in other parser generators (in fact, XText uses the well known ANTLR generator to create the actual parsing code). Some peculiarities of the employed grammar model are illustrated by example rules from Fig. 2.

As we can see, Fig. 2 shows a fairly standard example of an attributive grammar in EBNF notation. Attribute values can be obtained from tokens or rule

```
Function:
        "function" name=ID "(" arguments=ArgumentList ")"
            ":" resultType = Type
            ((preconditions+=Precondition) |
             (postconditions+=Postcondition) |
             (locals+=Locals))* body = HoareTriple;

Statement: SimpleStatement | CompositeStatement;
SimpleStatement: Skip | Assignment;
CompositeStatement: Block | Cond | Loop;

AssertionList: {AssertionList} (=> assertions += Assertion
    (=> assertions += Assertion)*)? ;

HoareTriple: precondition = AssertionList
                 statement = Statement
                 postcondition = AssertionList (=> ";")? ;

Call returns Expression: Var |
  {Call} function=[Callable] => "(" args=ActualArgs ")";
```

Fig. 2. XText grammar

calls. It is not possible to transform said values in any way, save by composing them into lists with the operator `+=`. Semantic predicates can be employed to disambiguate conflicts in the resulting $LL(*)$ parser. This is a convenient solution for, among other things, the dangling else problem.

One aspect of the presented grammar that requires a more detailed explanation is the presence of nonterminals enclosed in square brackets. This construct is used to represent references to other nodes of the syntax tree. A simple example is the call rule, which contains a reference to a procedure declaration. Syntactically, a reference corresponds to an identifier, that is, the `ID` lexical symbol (it is possible to use other symbols). The process of converting identifiers to node pointers is governed by scoping rules, which must be implemented in Java code in a manner which we now discuss.

Rules have the form of methods in a Java class. Their names and argument types are used to decide which rules apply to which reference. This technique is, unfortunately, rather error-prone, as there is no easy way to detect misspelled names. Alternative approaches have been developed, but, due to the simplicity of our project, we have not taken advantage of them. The example code in Fig. 3 shows some of the rules responsible for the visibility of local variables. First method ensures that, when a reference to local variable is being untangled inside a procedure, all declared locals will be available. Second simply adds a quantified variable to the environment used in the body of a quantified formula. The final rule states that a reference to a variable might point to a local, an

argument, or the function's result. An interesting aspect of the scoping mechanism is that it produces all possible references in given program point, instead of just searching for definition of a particular identifier. This allows XText to offer automated completion proposals (note that it is possible to further customize this mechanism).

```
IScope scope_Local(final Function function,
                   final EReference ref) {
  final List<IEObjectDescription> result =
      new ArrayList<IEObjectDescription>();
  for (final Locals locals : function.getLocals()) {
    for (final Local local : locals.getVariables()) {
      result.add(EObjectDescription.create(
              local.getName(), local));
  }}
  return new SimpleScope(IScope.NULLSCOPE, result);
}

IScope scope_Variable(final Forall forall,
                      final EReference ref) {
  return new SimpleScope(getScope(forall.eContainer(), ref),
    scope_QuantifiedVar(forall, ref).getAllElements());
}

IScope scope_Variable(final Function func,
                      final EReference ref) {
  final IEObjectDescription resultVar =
      EObjectDescription.create(func.getName(),
                                func.resultVar());
  return new SingletonScope(resultVar,
    new CompositeScope(scope_Local(func, ref),
        scope_Argument(func, ref),
        getScope(func.eContainer(), ref)));
}
```

Fig. 3. Scoping example

A parsed XText program is passed for further processing as an EMF model. EMF [25] is a vastly simplified variant of UML that is used thorough many Eclipse components, especially since Eclipse 4. Structure of a model is described by a so-called meta-model. It can be inferred from the XText grammar, but may also be provided separately. The latter option allows some customisation, such as adding attributes or methods not present in the grammar. Attributes of a model element are available in Java code by the use of accessor methods. This mechanism is used by the sample scoping code in Fig. 3. It is also possible to use reflection, and a switch class, which is a simple implementation of the visitor

pattern, generated by EMF for each metamodel. We use that class extensively for traversing programs, in particular during verification condition generation.

Parsing process ensures the absence of syntax errors in the input code. Scoping checks for issues with invalid references. But there are many other rules, of a more semantical nature, which must be satisfied by a program. For example, we might not want to allow quantifiers to be used in actual program expressions, and instead allow them only in assertions and other parts of specification. This particular condition could, in theory, be ensured by making changes to the grammar, but such approach is extremely cumbersome and results in a lot of clutter and duplication in the grammar. XText contains a mechanism which allows us to resolve this issue in a much more efficient and elegant manner. It is based on validators, which are simply Java classes in which some methods are marked with the `Check` annotation. Such methods are automatically invoked for each AST node that matches their argument type. Example in Fig. 4 shows the implementation of a few validator checks, including the aforementioned rule regarding quantifier use. It is important to note the way in which errors are reported. Methods used for this purpose are provided by XText, and their arguments include the precise location of the offending AST element. This allows the editor to link markers and other error indicators with the relevant part of the source code.

```
@Check public void checkForallType(final Quantifier expr) {
  for (final QuantifiedVar variable : expr.getVars()) {
    if (! (variable.getType() instanceof SimpleType)) {
      error("Quantification over this type is not allowed",
            variable, Literals.DECLARED_VARIABLE__TYPE, -1);
}}}
@Check public void
checkProgramRestrictionsForAssignment(final Assignment a) {
  checkExprRestrictions(a.getValue());
}
private void checkExprRestrictions(final Expression expr) {
  final Iterator<EObject> iterator =  expr.eAllContents();
  while (iterator.hasNext()) {
    final EObject sub = iterator.next();
    if (sub instanceof Forall || sub instanceof Exists) {
      error("Quantifiers are not allowed in expressions.",
        sub.eContainer(), sub.eContainingFeature(), -1);
    } else if (subexpr instanceof Call) {
      if (((Call)sub).getFunction() instanceof Predicate) {
        error("Predicates are not allowed in expressions",
          call, Literals.CALL__FUNCTION, -1);
}}}}
```

Fig. 4. Validation example

Let us conclude our discussion of the implementation of HAHA with a presentation of its core component, which is the verification conditions generator. Its implementation is surprisingly simple and clear. Rules of Hoare logic are mapped to methods of EMF switches, which are then used to traverse the program that is to be verified. We show one such method in full, to illustrate its simplicity. Each method takes an AST node as argument, and can access relevant pre- and post-conditions stored in the *preconditions* and *postconditions* fields of the switch object. One thing that should be noted about the way in which we generate the verification conditions is that we keep track of context, that is, the program elements associated with each condition and reasons why it was generated. This results in a slightly more verbose code, but is of immense use during verification. That is because the context is necessary to generate accurate error markers in case of prover failure. We also use it to store information about variables, predicates and axioms that should be visible inside a verification condition.

The method we present is concerned with processing of the conditional statements. The implementation is somewhat verbose. In spite of this, it can be clearly seen how the Java source corresponds to typical Hoare rule for conditional statements found in textbooks or other learning resources. As is to be expected, it is somewhat cluttered by code necessary to manage contexts and handle corner cases, such as conditionals with missing `else` part.

The discussed code is part of a visitor class and is supposed to compute the verification conditions and pass them to a consumer object. First few lines, shown in Fig. 5, are responsible for constructing a context object, which provides a link between the generated formulae and their source (an AST node). They also contain invocation of a function which converts the condition from an AST node to a first order formula.

```
public Boolean caseCond(final Cond cond) {
  final VCContext ifContext =
    new VCContext("cond_L" + getLine(cond),
                  ImmutableList.of((EObject)cond),
          "'If' at line " + getLine(cond), parentContext);
  final DocumentedFormula condFormula =
      exprToFormula.apply(cond.getCondition());
```

Fig. 5. Hoare rule for conditional statements - context construction

Conditional statements consist of a positive and negative part. Processing the first one is straightforward, as can be seen in Fig. 6. Condition expression is added to the list of preconditions and statements from the `if` body are processed recursively.

```
final VCContext trueContext =
    new VCContext("ifTrue",
            ImmutableList.of((EObject)cond.getIfTrue()),
        "Case 1 - condition holds.", ifContext);
List<DocumentedFormula> extendedPreconditions =
    new ArrayList<DocumentedFormula>(preconditions);
extendedPreconditions.add(condFormula);

generateVerificationConditions(
    trueContext, extendedPreconditions,
    cond.getIfTrue(), postconditions, consumer);
```

Fig. 6. Conditional statements: positive case

The negative case is somewhat more involved. To handle it, the condition formula must first be negated and added to the list of preconditions. This is done by the code in Fig. 7.

```
final DocumentedFormula negatedCondition =
    new DocumentedFormula(
      new Negation(condFormula.getFormula()),
      "negated_condition", ImmutableList.of((EObject)cond),
      "Negated condition");

extendedPreconditions =
    new ArrayList<DocumentedFormula>(preconditions);
extendedPreconditions.add(negatedCondition);
```

Fig. 7. Adding negated condition to the list of preconditions

In our grammar, the `else` part of a conditional statement is optional. This must be taken into account when context for verification conditions for the negative case is constructed. Fig. 8 shows how this issue is handled.

```
if (cond.getIfFalse() != null) {
  falseSource = cond.getIfFalse();
} else { falseSource = cond; }
final VCContext falseContext = new VCContext("ifFalse",
    ImmutableList.of(falseSource),
    "Case 2 - condition does not hold.", ifContext);
```

Fig. 8. Conditional statements: negative case context construction

Final part of the code, presented in Fig. 9, produces verification conditions for the case when the `if` condition is false. If there was an `else` part, it is processed recursively. Otherwise it is stated that the preconditions should imply the postconditions.

```
  if (cond.getIfFalse() != null) {
    generateVerificationConditions(falseContext,
            extendedPreconditions, cond.getIfFalse(),
                postconditions, consumer);
  } else {
    for (final DocumentedFormula post : postconditions) {
      final VerificationCondition vc =
            new VerificationCondition(
          extendedPreconditions, post, falseContext);
      consumer.processVerificationCondition(vc);
  }}
  return true;
}
```

Fig. 9. Hoare rule for conditional statements - conditions for the negative case

4 SMT solvers

We have seen how XText allows us to create an advanced and feature rich editor for Hoare programs. Yet such an editor would be of truly limited use if it was not capable of ascertaining the validity of the generated verification conditions in an automated manner. This crucial task is handled by satisfiability modulo theorem (SMT [4]) solvers. SMT (more precisely, SMT-LIB 2.0) is a standard which describes the input and output languages of such tools. It should be noted that solvers, as their name implies, are used to find a valuation that makes a formula true. Fortunately, modern solvers are also capable of verifying that a formula is not satisfiable, and so can be used as provers (since a formula is true iff its negation is not satisfiable). The model finding capability remains very useful as it is used to produce counterexamples when the verification conditions are not valid.

HAHA uses pipes to communicate with solvers via their standard input and output channels. This approach can be argued to be somewhat inefficient, but that is hardly a noticeable problem when the size of the input program, and number of generated formulae, is as small as it is in our case. Unfortunately, the whole mechanism used to communicate with solvers required a rather significant amount of code, especially compared to other, more interesting, modules, such as the verification conditions generator. We hope that this situation will soon be improved, and that a generic interface to SMT solvers will be made available

for Eclipse plugins (such as HAHA). When we undertook the development of HAHA, we were unable to find such a library with acceptable licence and set of features.

The area of automated satisfiability solving has enjoyed significant progress in recent years. The class of formulae that can be successfully processed is expanding. As an example, consider the sample program from Fig. 1. It contains conditions with nonlinear expressions. Hence simpler solvers, typically based on Presburger arithmetic, are not able to process the verification conditions generated for the discussed program. But more advanced tools, such as Microsoft Z3[12], have such capability. Still, there are cases, often involving exponentiation or division operators, which cannot be handled by any currently available solver. It is, however, possible to simplify the task by providing additional axioms. These are often augmented by instantiation patterns[23,13], which are used to decide when to use an axiom or precondition which contains quantifiers.

5 Related work

KeY-Hoare [8] is a tool that serves purposes very similar to HAHA. It uses a variant of Hoare logic with explicit state updates which allows one to reason about correctness of a program by means of symbolic forward execution. In contrast, the assignment rule in more traditional Hoare logics requires backwards reasoning, which can be argued to be less natural and harder to learn. Implementation of the system is based on a modification of the KeY [5] tool.

Why3 [6,15] is a platform for deductive program verification based on the WhyML language. It allows computed verification condtions to be processed using a variety of provers, including SMT-based solvers and Coq. WhyML serves as an intermediate language in verifiers for C [7,10], Ada [20] and Java. It has also been used in a few courses on formal verification and certified program construction.

Another tool used in education that must be mentioned here is Dafny [21]. It can be used to verify functional correctness and termination of sequential, imperative programs with some advanced constructs, such as classes and frame conditions. Input programs are translated to language of the Boogie verifier, which uses Z3 to automatically discharge proof obligations.

Some courses on formal semantics and verification use the Coq interactive theorem prover as a teaching aid [24,17]. Reported results of this approach are quite promising, but the inherent complexity of a general purpose proof assistant appears to be a major obstacle [24]. One method that has been proposed to alleviate this issue is to use Coq as a basis of multiple lectures on subjects ranging from basic propositional logic to Hoare logic [17]. In this way the overhead necessary to learn to effectively use Coq, or a similar tool, becomes less prominent.

HAHA avoids problems with steep learning curve of interactive provers by using an automated solver. It should be noted, however, that having the ability to examine and experiment with the proof obligations in a tool such as Coq can

be beneficial for users already familiar with said tool. For this reason, we have implemented a command which makes it possible to produce Coq formulae from generated verification conditions.

The complexity of general purpose provers is often a troublesome issue in education. One can attempts to resolve this problem by creating tools tailored to specific applications, which sacrifice generality for ease of use. One example of such a system is SASyLF [1], which is a proof assistant used for teaching language theory. Another program worth mentioning here is CalcCheck [19]. It is used to check validity of calculational proofs written in the style of a popular textbook [16]. This approach is very similar to what we advocate for teaching Haore logic, as it shows an example of a tool created to fit the style of existing educational material.

6 Conclusions

We hope that, even though our presentation of HAHA was rather brief, it was a convincing argument in support of our thesis. While there is certainly a lot of further work to be done, we have already achieved main goals stated when we started the HAHA project. The tool has been implemented, and was presented to some of the students participating in the course on semantics and verification of programs at the University of Warsaw. Survey conducted among those students after the final exam showed that our tool is a welcome and useful aid in learning Hoare logic (there was, of course, also some criticism, especially considering the documentation, as well as the heavyweight nature of the Eclipse platform).

Let us note that XText and Z3 are not the only tools available to teams which might endeavour to build a formal verification tool for use in teaching. We have chosen these technologies as they appeared to best suit our needs and technical expertise, but there are many alternatives. All major development environments, such as IDEA, Netbeans or Sharpdevelop, have frameworks for creating domain specific languages. The use of simple source highlighting components, such as Scintilla, is also worth investigation. Creating a full fledged editor with all required features is likely to require significantly more coding, but this is balanced by increased flexibility and lack of overhead caused by layering multiple frameworks, which is very typical for platforms such as Eclipse. This last issue proved to be quite inconvenient during HAHA development, so it is plausible that an approach that is free of it might prove to be successful. Regarding the solver technology, one could try to use CVC4 [3], or even try to implement a specialized solver, which is a very interesting project in itself.

References

1. J. Aldrich, R. J. Simmons, and K. Shin. Sasylf: an educational proof assistant for language theory. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education*, FDPE '08, pages 31–40, New York, NY, USA, 2008. ACM.

2. K. R. Apt. Ten years of hoare's logic. *ACM Transactions on Programming Languages and Systems*, 3:431–483, 1981.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
4. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
5. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
6. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
7. S. Boldo and C. Marché. Formal verification of numerical programs: From C annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5(4):377–393, Dec. 2011.
8. R. Bubel and R. Hähnle. A hoare-style calculus with explicit state updates. In Z. Instenes, editor, *Proc. Formal Methods in Computer Science Education (FORMED)*, Electronic Notes in Theoretical Computer Science, pages 49–60. Elsevier, 2008.
9. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
10. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.
11. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE Companion*, pages 429–430. IEEE, 2009.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.
13. Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.
14. M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *SPLASH/OOPSLA Companion*, pages 307–309. ACM, 2010.
15. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
16. D. Gries and F. B. Schneider. *A logical approach to discrete math*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
17. M. Henz and A. Hobor. Teaching experience: Logic and formal methods with coq. In *CPP*, pages 199–215, 2011.

18. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press.
19. W. Kahl. The teaching tool calccheck: a proof-checker for gries and schneider's “logical approach to discrete math”. In *Proceedings of the First international conference on Certified Programs and Proofs*, CPP'11, pages 216–230, Berlin, Heidelberg, 2011. Springer-Verlag.
20. J. Kanig. Leading-edge ada verification technologies: combining testing and verification with gnattest and gnatprove – the hi-lite project. In *Proceedings of the 2012 ACM conference on High integrity language technology*, HILT '12, pages 5–6, New York, NY, USA, 2012. ACM.
21. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.
22. K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, Oct. 2000.
23. D. Luckham, S. German, F. von Henke, R. Karp, P. Milne, D. C. Oppen, W. Polak, and W. Scherlis. Stanford pascal verifier user's manual. Technical Report CS-79-731, Department of Computer Science, Stanford University, Stanford, CA, 1979.
24. B. C. Pierce. Lambda, the ultimate ta: using a proof assistant to teach programming language foundations. In *ICFP*, pages 121–122, 2009.
25. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2 edition, 2009.