Deductive Verification Tool Review

Jianjie Yan - 16348301
Department of Computer Science
National University of Ireland, Maynooth
Jian.yan.2017@mumail.ie

Overview of the tools

Dafny

Dafny [1] is an open-source verification tool designed by K. Rustan M. Leino at Microsoft Research. Dafny is a programming language, verifier, and compiler. Dafny supports formal specification through preconditions, postconditions, loop invariants and loop variants. Dafny builds on an intermediate language Boogie which uses the SMT solver to automatically prove correctness.

Dafny has its own language, it is similar to Java and C# with built-in specification constructs. The input file of Dafny will contain the Dafny source code with the specifications like post/pre-conditions. The input code is converted into C# code, and then convert in CIL code in the end. From verified programs, the Dafny compiler produces code (.dll or .exe) for the .NET platform.

OpenJML

OpenJML [2] is developed by David Cok. It is based on JML, the OpenJDK compiler, Eclipse plug-ins, and SMT-lib based SMT solvers. The Java Modelling Language (JML) [3] is a specification language for Java program using Hoare style specifications.

OpenJML is a suite of tools for editing, parsing, type-checking, verifying (static checking), and run-time checking. The annotated JML statements of Java programs states that what the methods inside of the program is supposed to do and the invariants the data structures should obey, JML annotations state preconditions, postconditions, invariants of the class or method. OpenJML's tools will then check that the implementation and the specifications are consistent.

Since the tool is built based on JML and OpenJDK compiler. The input and output for OpenJML are the standard Java input and output with JML annotations added.

How Do They Verify Programs?

OpenJML

OpenJML was constructed by extending OpenJDK [4], "*the open-source Java complier to parse and include JML constructs in the abstract syntax trees created by the Java compiler to represent the Java Program*" [5].

The design and implementation of OpenJML uses and extends many ideas present in prior tools, such as ESC/Java2 and JML. "*It transforms a JML-annotated program into a static single assignment form, and then generates first-order logic verification conditions from this transformed program. This output format is suitable for a satisfiability modulo theory (SMT) solver*" [6].

The SMT solver are not part of OpenJML itself. OpenJML needs to be told which solver to use and the location of its executable in the local system, the default SMT solver is Z3[7]

Dafny

The Dafny programming language is object-based, imperative, sequential, and supports generic classes and dynamical location. The specification constructs include standard pre- and postconditions, framing constructs, and termination metrics, these specifications are based on Hoare Logic. Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie [9], the output from Boogies is then translated back to Dafny. Thus, the semantics of Dafny is defined in terms of Boogie. Boogie is a layer on which to build program verifiers for other languages [10].

Architecture: Software Components of the tools

Dafny

Dafny has a VS extension, when user editing the program, the VS extension sends the change to underlying Dafny verifier, and as discussed above, the code is translating into the intermediate verification language Boogie for the verification. The SMT solver used by Boogie is Z3. The verification result is then sent back to Dafny by translating the Boogie language back to Dafny.

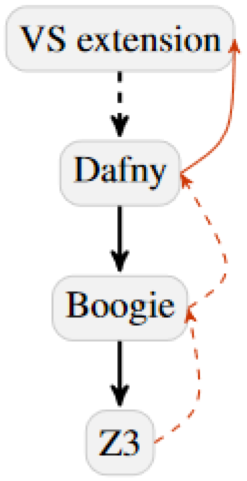


Figure 1: Dafny Architecture [10]

OpenJML

OpenJML is a tool built on the OpenJDK Java compiler. The architecture of OpenJDK and OpenJML is shown in Figure.2. The OpenJDK has multiple phases, source code inputted is going through Scanning, Parsing, Name Resolution and Type-checking phases, producing a forest of Abstract Syntax Trees (ASTs) representing the program. This AST is then going through a series of code generation and then Java byte code is produced. For OpenJML, the Java classes are extended by JML versions. Scan, parse, name resolve, and type-check are text contained in the JML annotations along with the Java code. The JML annotations are converted into assumptions and assertions that are inserted into AST. This translation step embodies the semantics of the JML specification. The modified ASTs can be sent to the (unmodified) code-generation phase to produce output bytecode with embedded JML assertions. Or the ASTs can be sent to the Java and JML logical encoding phase, which produces an SMT. SMT solver can then determine whether all assertions in the SMT encoding are valid. A central point of this architecture is that the JML semantics are embodied in the JML-enhanced AST, which is used by both runtime and static checking [6].

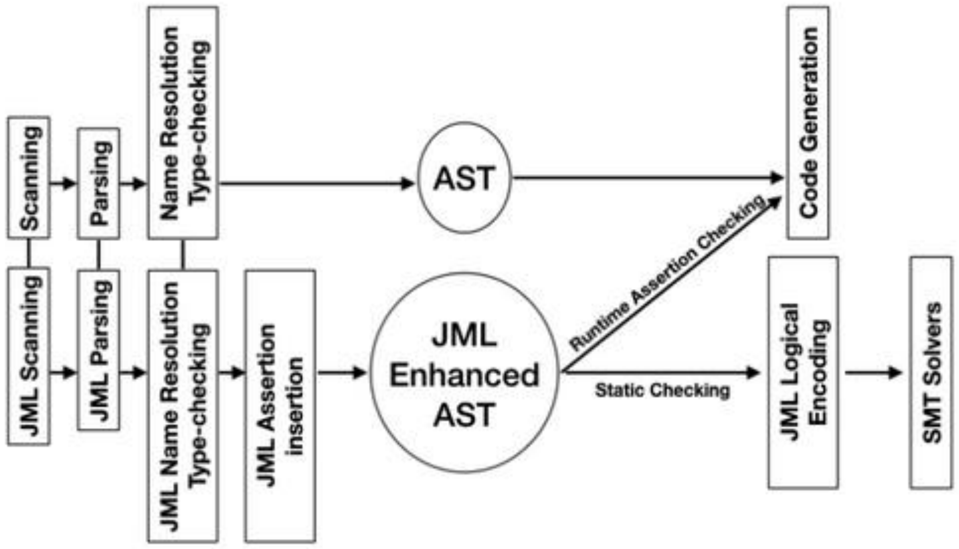


Figure.2: Architecture of OpenJDK and OpenJML [11].

Type of Programs That the System Can Verify

Dafny

As discussed in the before, Dafny language is similar to Java and C# and it can be converted into C# when compiling. Dafny language has built-in specification constructs, the specifications include pre/post-conditions, frame specifications, and termination metrics. Dafny language has many cluses for these specifications, pre/post-conditions can be declared with 'requires' and 'ensures' clauses. Assertions is declared with keyword 'assert', and some other keywords for other specifications.

Dafny also has support ghost variables, recursive functions and types like sets and sequences. Specifications and ghost constructs are used only during verification; the compile omits them from the executable code.

OpenJML

The language that OpenJML can verify is Java, with JML annotations embodied. OpenJML requires the class to be compilable due to OpenJML's use of OpenJDK to produce ASTs. In OpenJML, specifications are written using "//@" and the keywords such as 'requires' and "*ensures"*. JML also support ghost variable and loop invariant.

```
//@ requires "Specifications"
//@ ensures "Specifications"
```

Type of Errors

OpenJML

OpenJML requires a compilable class, so it can immediately identify programs errors without spending a lot of time on the verification of a non-compilable program. OpenJML checks for bounds within the preconditions. Two things about errors in contracts are stated by JML standard: First, it states that statements in a contract are to be evaluated in order, such that certain exceptions are be prevented by verifying that the index is within bounds. Second, it states that a condition is valid if it evaluated to true.

"*OpenJML supports two verification approaches: runtime verification finds errors when they happen but does not verify the correctness of all possible programs, while static verification aims to prove correctness of all executions but might indicate errors that will never happen during an execution*" [13].

Dafny

There are two main causes for Dafny verification errors: annotations that are inconsistent with the code, and situations where it is not "*clever*" enough to prove the required properties.

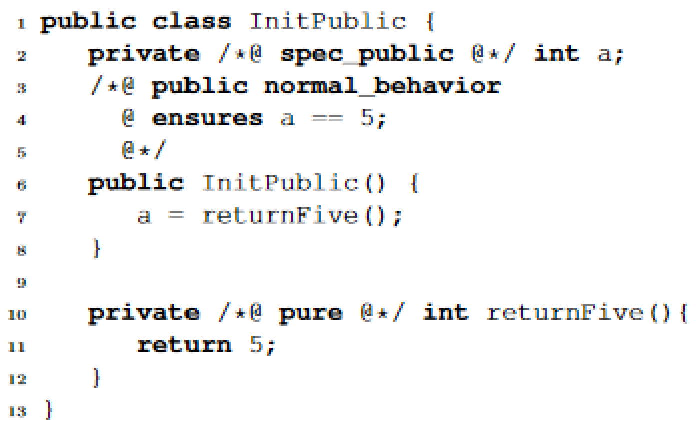
Dafny proves that there are no run time errors, such as index out of bounds, null dereferences division by zero, etc. for user supplied annotations. It also proves that termination of code, except in specially designed loops [14].

Strengths and Limitations

OpenJML

OpenJML can immediately identifies program errors but does not make it easy to verify single classes in isolation. In OpenJML, user often has to spend a lot of time on stripping irrelevant imports, function calls etc., in order to make the verification working.

OpenJML checks for visibility, it can report all visibility issues in the given specification. OpenJML uses a very puristic approach: any method call will be abstracted by its method specification. The method ReturnFive in figure 3 has no method specification. OpenJML will simply assume that any behavior of this method call is possible, and sometimes will not be able to prove the postcondition a==5. [13]



```
public class InitPublic {
   private /*@ spec_public @*/ int a;
   /*@ public normal_behavior
     @ ensures a == 5;
     @*/
   public InitPublic() {
      a = returnFive();
   }

   private /*@ pure @*/ int returnFive(){
      return 5;
   }
}
```

Figure.3: Method without a contract [12]

Dafny

When Dafny verify a program, it “*forgets*” about the body of every method except the one it is currently working on, so that we can analyse each method separately. Dafny doesn’t care what happens inside of methods, it only cares if the annotations are satisfied. This property of Dafny increases the operating speed of Dafny.

Dafny allows programmer to include ghost variable in verification without affecting the performance of the executable program itself. This property is also true is OpenJML.

Predicates or lemmas are not supported in the body of the postconditions although they can help the modular for the specification of the programs.

Ensures is only supported for postconditions in function contracts, it would be helpful if we can introduce ensures in other places such as invariant declarations.

Examples of Two Systems

Let’s take a loop of some examples for the two systems.

Example1:
Find the Max and Sum of array.
OpenJML:
The method m finds the max and sum of input array a.

```
public class SumMax {
  //@ requires a.length > 0;
  void m(int[] a) {
    int sum = 0;
    int max = a[0];
    //@ loop_invariant 0 <= i <= a.length;
    //@ loop_invariant sum <= \count * max;  // Assertion to be proved
    for (int i=0; i<a.length; i++) {
      //@ assume Integer.MIN_VALUE <= sum + a[i] <= Integer.MAX_VALUE;
// Just assume we never overflow
      sum += a[i];
      if (max < a[i]) max = a[i];
    }
  }

}
```

As we can see from the code above, the class contains simple Java code with some JML annotations added.

The method m takes an input array, a precondition “*//@ requires a.length > 0*;” is specified with “*requires*” keyword. OpenJML also support loop invariant to handle the loop in Java, loop invariants are what must be true before and after the loop, first loop invariant states that 0<= i <= a. length, which is true before and after the loop. We can also add assumptions to the code by using ‘assume’ keyword.

Dafny

This method takes an int N (Length of Array) and an int array, it returns the max and sum of the array.

```
method M(N: int, a: array<int>) returns (sum: int, max: int)
  requires 0 <= N && a.Length == N && (forall k :: 0 <= k && k < N ==>
0 <= a[k]);
  ensures sum <= N * max;
{
  sum := 0;
  max := 0;
  var i := 0;
  while (i < N)
    decreases N-i
    invariant i <= N && sum <= i * max;
  {
    if (max < a[i]) {
      max := a[i];
    }
    sum := sum + a[i];
    i := i + 1;
  }
}
```

Dafny support its own language, but similar to Java and C#. Dafny allows programmer to add the specifications without using "//@" like OpenJML, which makes it convenience when writing specifications.
Dafny support quantifiers, in this case, forall is a quantifier. It means that for all k from 0 to N, 0<=a[k].

Dafny requires decreases clause when it comes to loop, and recursive functions and method. A decreases annotation specifies a value, called the termination measure, that becomes strictly smaller each time a loop is traversed or each time a recursive function or method is called and the value decreases to zero.

Example2:

In this example, we demonstrate the how to verify a program that finds the maximum number of an array in two systems.

OpenJML:

```java
public class MaxByElimination {

  //@ requires a != null && a.length > 0;
  //@ ensures 0 <= \result < a.length;
  //@ ensures (\forall int i; 0 <= i < a.length; a[i] <= a[\result]);
  public static int max(int[] a) {
    int x = 0;
    int y = a.length-1;

    //@ loop_invariant 0 <= x <= y < a.length;
    // So far either a[y] is the largest or a[x] is the largest of ever
ything beyond x and beyond y (not including a[x] and a[y])
    /*@ loop_invariant ((\forall int i; 0<=i && i<x; a[i] <= a[y]) && (
\forall int i; y < i && i < a.length; a[i] <= a[y]))
                 ||  ((\forall int i; 0<=i && i<x; a[i] <= a[x]) && (\f
orall int i; y < i && i < a.length; a[i] <= a[x]));
     */
    //@ decreases y-x;
    while (x != y) {
      if (a[x] <= a[y]) x++;
      else y--;
    }
    return x;
  }
}
```

In this example, we introduce a new specification “*ensures*”. This is the keyword used to specify the postcondition of the method. Since we are writing in Java, there is no return variable declared before the method, we use “*\result*” for the return value. This is different from Dafny, Dafny specifies the return variable before method body, so programmer in Dafny can use the declared variable straightly when writing pre/post-conditions.

OpenJML also support quantifier, there are two types of quantifiers, forall and exists. We only use forall in this example.

OpenJML also has decreases clause, same as Dafny, decreases clause is a value that gets smaller after a loop, and finally decreases to 0.

Dafny

```
method Maximum(values: seq<int>) returns (max: int)
  requires values != []
  ensures max in values
  ensures forall i | 0 <= i < |values| :: values[i] <= max
{
  max := values[0];
  var idx := 0;
  while (idx < |values|)
    invariant max in values
    invariant idx <= |values|
    invariant forall j | 0 <= j < idx :: values[j] <= max
  {
    if (values[idx] > max) {
      max := values[idx];
    }
    idx := idx + 1;
  }
}

lemma MaximumIsUnique(values: seq<int>, m1: int, m2: int)
  requires m1 in values && forall i | 0 <= i < |values| :: values[i] <=
 m1
  requires m2 in values && forall i | 0 <= i < |values| :: values[i] <=
 m2
  ensures m1 == m2 {
    // This lemma does not need a body: Dafny is able to prove it corre
ct entirely automatically.
}
```

OpenJML requires a compilable class for the verification, where Dafny can verify a single method. We can see a keyword “*in*” from the first postcondition, this is a short way of saying max is inside of array values.

Dafny supports lemma, a lemma is like a ghost method without a body, it is declared as the postconditions, Dafny can prove the correctness automatically. Lemma doesn’t need to be called at run time and the compiler erases it before producing executable code.

Uses of the tools.

Dafny
Dafny was widely used in teaching area to provide students simple introduction to formal specification and verification. Dafny is also commonly used in verification competitions (e.g., VERIFYTHIS, VSCOMP).

OpenJML
OpenJML is a tool used to verify Java programs with JML specifications, the goal of OpenJML is to implement a full tool for JML and current Java that is easy for practioners and students to use to specify and verify Java programs.
It is widely used in teaching area, there is a Verily based web application for learning about the Java Modelling Language called TryOpenJML is created by a Tushar Deshpande from University of Central Florida. Which helps people understand JML and can test and evaluate Verily and its features.

Conclusion.

We can install Dafny on VS code and OpenJML can be installed as Eclipse plug-in or as command line tool. Both OpenJML and Dafny provide static verification, and OpenJML also provide run-time verification. Static verification aims to prove correctness of all executions but might indicate errors that will never happen during an execution. Runtime verification finds errors when they happen but does not verify the correctness of all possible programs.

OpenJML is implemented based on OpenJDK and JML, so OpenJML can verify compilable Java programs. Dafny has its own language, but it is easy to learn because of its similarity to Java and C#.

Reference.

[1]: Open-source project, "*Dafny*". Available: https://github.com/dafny-lang/dafny/blob/master/README.md
[2]: Website documentation. *"OpenJML - formal methods tool for Java and the Java Modeling Language (JML)*". Available: https://www.openjml.org/documentation/introduction.html
[3]: Website Documentation,
"*The Java Modeling Language (JML)*" https://www.cs.ucf.edu/~leavens/JML/index.shtml
[4]: "OpenJDK Documentation". Available: https://openjdk.java.net/

[5]:David R.Cok, “*OpenJMLUserGuide.pdf 2018*”. Available: https://github.com/OpenJML/OpenJML/blob/master/OpenJMLUI/html/OpenJMLUserGuide.pdf
[6]: Jan Boerman, Marieke Huisman, and Sebastiaan Joosten, “*Reasoning about JML: Difference between Key and OpenJML*”, August-2018.
Available: https://wwwhome.ewi.utwente.nl/~marieke/KeyOpenJML.pdf
[7]: Open-source Z3 tool. Available: https://github.com/Z3Prover/z3
[8]: Open-source Boogie Verifier tool. Available: https://github.com/boogie-org/boogie
[9]: Paqui Lucio, “*A Tutorial on Using Dafny to Construct Verified Software*”, published 2017.
Available: https://arxiv.org/pdf/1701.04481.pdf
[10]: https://arxiv.org/pdf/1404.6602.pdf
[11]: K. Rustan M. Leino, Valentin Wüstholz, “*The Dafny Integrated Development Environment*”, published 2014. Available: https://www.researchgate.net/figure/Architecture-of-OpenJDK-and-OpenJML_fig1_328597655
[12]: Jason KOENIGaand K. Rustan M. LEINO[b] at Microsoft Research. “*Getting started with Dafny: A Guide*”, published 2012. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml220.pdf