Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Veri-J – An IntelliJ Plugin for Java Verification

Author:
Rohan Khosla

Supervisor:
Dr Mark Wheelhouse

Second Marker:
Dr Dalal Alrajeh

March 31, 2025

Abstract

Computing students at Imperial learn formal methods for software verification during their first year. The module concerned is theoretical but the course tutors would also like to have a software tool that allows students to practice what they learn. Using an existing software verification tool for this would be unsuitable as the students have little time to learn a new language alongside their existing work. Therefore, a custom tool was required, which students could use with their existing Java programming skills and the proof-writing syntax that they learn on the course.

Veri-J is a plugin for the IntelliJ IDE that has been built for this purpose as an MEng Computing final project. It accepts all the Java and specification constructs used on the module. It translates these into the Dafny specialist program verification language, submits them via Dafny to the Z3 SAT/SMT solver and displays verification results from Z3 to the user. It includes error handling for syntax, semantic and verification errors, which displays details of the error to users and highlights error locations in the source file. Performance tests demonstrate that it responds within 2-3 seconds, which will allow students to build their proofs interactively, adding or changing input step-by-step in response to the feedback they receive.

Acknowledgements

I would like to thank Dr Mark Wheelhouse, my supervisor, for his guidance throughout this project. Mark was always accessible and unstinting with his time. His advice and comments during our fortnightly online meetings and on the draft of this report have been invaluable. I would also like to thank my family and friends for their support during my degree at Imperial.

Contents

Chapter 1

Introduction

1.1 Context

Achieving mathematical guarantees of correctness for any piece of software is incredibly demanding. However, despite the costs, achieving such certainty is becoming more and more worthwhile. Mission- and safety-critical systems, such as in avionics, power transmission or medical devices are increasingly pervasive in modern society. This trend can only accelerate as automation continues rapidly to transform a growing number of industries. At the same time, many critical sectors of the economy such as social media and life sciences are generating vast quantities of data which in turn is giving rise to an explosion of software systems that are used to inform key decisions that significantly affect almost every aspect of our lives.

These advances pose several challenges for software and hardware engineers:

- How does one build safety- and mission-critical software, in any domain?
- What are the techniques that need to be employed?
- How can these techniques be integrated within the conventional software engineering process to minimise the time and cost involved?

Conventionally, these questions have been addressed through software testing. Techniques such as 'fuzzing', property-based testing and even step-by-step debugging of code have, and continue to be, used. However, as often demonstrated by security breaches or episodes such as PayPal's crediting $92 quadrillion to a user[1], software testing is unable to deliver absolute guarantees of correctness. Often, the final stage before production is simply to plan to monitor use and hope for the best.

Although testing remains essential, its limitations have motivated computer scientists to apply mathematical thinking to the specification, development and verification of hardware and software systems. The techniques developed in pursuit of this goal have collectively come to be known within the discipline as 'formal methods'. Formal methods connect a program's specification to its implementations in a firm and reliable way. The 'specification' is a clear and precise description of *what* the program has been written to do. It is usually defined using the language of formal logic or mathematics. The 'implementation' is usually the program itself and describes *how* the program achieves the goals set out in the specification. It is written in a programming language or pseudocode. Verification is the process of demonstrating with certainty that a program's implementation is equivalent to its specification. Provided the specification is free of errors, verification provides a mathematical guarantee that the implementation is correct.

1.2 Motivation

At Imperial College London, undergraduates are introduced to formal methods for software verification on the Discrete Maths, Logic and Reasoning (DML&R) module during the second term of their first year. Using standard algorithms, the students learn how to reason about correctness with preconditions, postconditions, loop invariants and lemmas. The objective is not only that the students should become familiar with verification but also that the experience of writing specifications should help to develop their understanding of how to write efficient, effective code. The module is theory intensive. Practice exercises are discussed during tutorials but are solved manually, on paper. The course faculty want to introduce a practical, hands-on element to the course to complement this theory-based approach. A software teaching tool is needed to achieve this.

One option would be to use the existing Dafny program verification language for this purpose. Dafny is a specialist imperative programming language with built-in support for inline specifications, which it verifies using the Z3 SMT/SAT solver. However, many if not most students on the course will have had little or no programming experience before joining Imperial. They will have been introduced to Haskell in their first term and will start learning Java in the second term. Having to learn another language like Dafny specifically for the course would increase their workload unacceptably. A custom-built tool is therefore being commissioned as an MEng Computing final-year project. The tool will be implemented as an interactive plug-in for IntelliJ Idea, which is the IDE used on the course. It should accept Java code with inline specifications in the format used on the course, translate this to Dafny, submit the translation to the Dafny verifier and return the result to the user with feedback relevant to the Java input code.

1.3 Objectives

The purpose of the project is to develop a software tool that students will use to practise the verification techniques that they learn on the Reasoning sub-module. The tool must make learning easier, otherwise students will see it as extra work and ignore it. The ambition is to create a learning aid that students will use to build proof solutions interactively, without their having to know anything more than they need to for the Reasoning sub-module itself. Each objective below addresses a quality that the software will need to achieve that ambition.

Full range of Java code

The plug-in should be able to accept and translate all the Java declarations, statements and expressions used on the module.

Same syntax for annotations

The plug-in should use the same syntax for specification annotations as used on the module. Students should be able to enter preconditions, postconditions, invariants etc using the same keywords and constructs as they do on paper. Symbols (such as $\forall$ and $\exists$), which are impracticable to type, should be replaced by keywords based on their spoken representation.

Comprehensive error handling

The plug-in should return error messages for syntactical, semantic and verification errors, in that order. Errors should be visually highlighted in the opened source file. The error message should state the nature and location of the error. Where practicable, it should also say what the user can do to correct it.

Intuitive functionality

Users should be able to start using the plugin independently to build proofs after an initial demonstration and/or reference to a user guide. They should need only the knowledge they acquire on the course to do this.

Integrated with IntelliJ

The plugin should use only standard IntelliJ and Java Swing components in the construction of its user interface. These should be laid out the in same way as in existing, well-established, IntelliJ plugins such as Git and Gradle. It should have the same formatting attributes, such as colour schemes and fonts, as the IntelliJ main window.

Easy to install and keep up to date

The tool should install at first attempt and update automatically using IntelliJ's standard facilities for doing so.

Extensibility

The plug-in should be extensible to accommodate changes in the notation used in the Reasoning sub-module. It should allow for future inclusion of any feature of Java that has an equivalent in Dafny and also of any feature of Dafny that is considered relevant to the needs of the Reasoning sub-module in future.

1.4 Contributions

This project has made five contributions:

- An IntelliJ plugin that verifies Java code annotated with proof statements. It has a simple, intuitive user interface that allows users to select a proof written in Java and determine its correctness (section 3.1). The plugin responds within 2-3 seconds for problems of the complexity encountered on the Reasoning sub-module (section 5.2.5).
- A syntax for proof statements that covers all the requirements of the Reasoning sub-module in a familiar notation. This includes expressions for quantifiers, comparison chains, array slicing and concatenation and input parameter entry-state values (section 3.4).
- A translator that translates Java code annotated with proof statements to the equivalent code in Dafny. The translator maps locations in the Dafny output to the corresponding locations in the Java input, to be used for verification error reporting (section 4.5).
- Error handling for syntax, semantic and verification errors that locates and highlights errors in the Java input code and displays error descriptions in the user interface (section 4.7).
- A comprehensive explanation of the principles of automated software verification, from conversion of code to proof formulae through to the most recent developments in SAT and SMT solvers (Chapter 2).

Chapter 2

Background

It can be challenging, even for experienced programmers, to write specifications and code that successfully verify using formal methods. The process of finding errors or omissions can be time consuming and, often, frustrating. At such times, the programmer needs to be confident that the verification tool they are using is giving them correct results or else they will stop using it. Such confidence is more likely to develop if the programmer understands, at least in principle, what is going on 'under the hood' of the verifier. The knowledge also creates awareness of the tool's capabilities and therefore of what it can or cannot be used to do.

This chapter explains the basic principles of automated software verification. It begins by outlining how temporal program code is converted to a static propositional logical formula. Next, it explores the inner working of a SAT solver, starting with simple backtracking and building up to the DPLL and CDCL algorithms. It considers the potential benefits of using CDCL for program verification instead of DPPL. After that, it explains the role of the SMT solver and how this interacts with the SAT solver to produce a result. Finally, there is an overview of the Dafny program verification language at the end of the chapter.

2.1 How logic is applied to program verification

The input to a SAT solver takes the form of a propositional logical formula. This is a combination of Boolean variables joined by the logical operators And, Or, Not and Implies. For example, a propositional logical formula involving four binary variables a, b, c and d could be:

$$(a \vee c \vee d) \wedge (a \vee b) \wedge (b \vee \neg d) \wedge (c \vee \neg d) \tag{2.1}$$

A key question to ask about such a formula is whether there is an assignment of true and false to some or all of its variables that makes the entire formula true, i.e., whether or not the formula is satisfiable. We can use this question to prove that a formula is *always* true simply by asking whether its negation is *ever* true. In other words, if there is no combination of values of the variables (a 'counter-example') that satisfies the complement of a formula, the formula itself must always be true and so has been proved.

The technique for deriving logical proof formulas from program code is known as symbolic execution. During normal ('concrete') execution, the variables in a program will be assigned different values, defined in terms of literals and/or each other. The core idea of symbolic execution is that if each such transient or 'temporal' value in a program path is uniquely assigned to a static symbol, then the relationships between these symbols form constraints which, taken together, constitute the logical proof formula of that path[2][3]. (Symbolic 'execution' is so named because, for practicability, the derivation is usually automated by executing an interpreter on the code to be verified.)

For example, if some variable x is assigned values five times during a program's execution, these values could be represented by the static symbols $x_0, x_1, \ldots, x_4$. Similarly, the values assigned to some other variable y could be represented by $y_0, y_1, \ldots, y_7$ and so on. Then, the logical proof formula for the relevant program path could be, say:

$$(x_0 == 3) \wedge \ldots \wedge (x_1 == y_3 + 7) \wedge \ldots \wedge (x_2 == x_1 * x_1) \wedge \ldots \wedge (y_0 == z_4 + x_3) \wedge \ldots etc., \quad (2.2)$$

to include all relationships between all values of all variables encountered in the program path. Pre and post conditions are simply added to the formula as additional constraints.

Many variable assignments take place in method and loop blocks. As these blocks execute multiple times, each assignment statement within them causes multiple changes to the value of the variable concerned. These repeated assignments need to be separated from each other so that the variable can be replaced by a new symbol each time its value changes. The symbolic interpreter does this using techniques known as in-lining of methods and unrolling of loops. After these, it performs the final stage, known as single static assignment (SSA)[4]. The three steps are described next.

2.1.1 In-lining of methods

If the code in a method can assign values to variables declared in the code that calls it, then such assignments need to be shifted into the calling code, where the variables concerned are replaced with symbols. This is done by replacing each method call with the code from the method's body:

For example, the code snippet:

```
fn(a, b)  {
    <body code>
}

fn(2, 3);
<more code>
```

would be replaced by:

```
a=2;
b=3;
<body code>
<more code>
```

As a specialist program verification language, Dafny has some built-in measures to limit the need for method in-lining. Firstly, all method input parameters are passed by value rather than by reference[1]. Therefore, any assignments made to these parameters in the method have no effect elsewhere[2], so the method does not need to be in-lined because of them. However, a method may change values held in reference memory using pointers passed to it as input parameters. Dafny requires such access to be declared in the method specification, using a 'modifies' statement[7], which tells the interpreter that the method needs to be in-lined.

2.1.2 Unrolling of loops

Variable assignments in loops are separated by converting the loop to nested If statements. Each such If statement repeats the code from the loop's body.

[1] This includes pointers to reference-type variables[5]

[2] Such assignments are, in fact, not possible because Dafny also makes input parameters immutable. However, this is to avoid user confusion rather than to limit method in-lining[6]. (The Veri-J plugin circumvents this restriction.)

For example, the loop:

```
while (x < 7) {
    <body code>
}
```

would become:

```
if (x<7) {
   <body code>
   if (x<7) {
      <body code>
      ...
   }
}
```

Early techniques for symbolic execution required the user to set an arbitrary bound for the number of unrolls. Current techniques use a loop variant and a loop invariant to set a precise bound on the number of iterations. The loop invariant is a Boolean condition that holds before and after each iteration. The variant is a mathematical function whose range usually is restricted to the non-negative integers. Provided the invariant holds, the variant decreases monotonically with each iteration of the loop, thereby setting an upper bound on the number of iterations before the loop terminates.

The invariant is defined within the loop specification. Current symbolic interpreters are usually able to determine the variant by themselves from the loop guard and body, though they do sometimes need it to be defined instead.

2.1.3 Single static assignment (SSA) of variable values

The replacement of functions and loops as described in the previous two steps allows SSA to take place. As mentioned above, in this step each variable in the program is given a new name or symbol for each time-step .

For example, the code:

```
x = 10;
y = 15;
x = x + y;
```

would become:

x_0 = 10;
y_0 = 15;
x_1 = x_0 + y_0;

Program branching adds complexity to SSA because the value to be assigned to a symbol after the branches have converged may be different depending on which branch was executed. As shown in the example below, the interpreter deals with this ambiguity by using a 'ϕ (phi) function' in such assignments. The ϕ-function can be thought of as a way for the interpreter to tell itself that it needs to submit alternative proof formulas to the SAT solver depending on the path condition, with each such formula having a different constraint for the symbol concerned.

For example, the code:

```
x = <some value>;
if (x < 7) {
    x = x + 1;
} else {
    x = x + 2;
}
```

would become :

```
x0 = <some value>;
if (x0 < 7) {
   x1 = x0 + 1;
} else {
   x2 = x0 + 2;
}
x3 = φ(x1, x2);
```

So if $x_0 < 7$ then the proof formula would include the constraint $(x_3 == x_1)$. But if $x_0 \geq 7$ the formula would include the constraint $(x_3 == x_2)$ instead.

Branching presents a challenge for symbolic execution because it increases the number of feasible programs paths (and hence the number of proof formulas) exponentially as program size increases. This problem, known as 'path explosion', makes it impracticable to symbolically execute all feasible program paths for large programs.

The techniques to mitigate path explosion include:

- Random methods to identify infeasible program paths, i.e., paths that are never executed and so do not need to be verified[8].

- Parallelisation. This technique identifies groups of independent paths, i.e., paths whose symbols are independent of each other, for verification in parallel.. Such groups can be identified by similarities in their path constraints. Each such group can be proved independently of the others by including preconditions that exclude variable values not relevant to it during concrete execution. The technique splits the symbolic model vertically along path length, with each group of paths then being verified in parallel by a different processor to reduce the overall verification time[9].

- Dynamic State merging. One way to reduce the number of program states that the solver needs to analyse is simply to merge similar path formulas. This often works but equally can increase processing time by increasing the number of disjoints in the combined formula. The solver may have to unravel the formula at these disjoints, which takes it longer than processing the original statements separately. In dynamic state merging, the interpreter first assesses candidate paths for this outcome and merges them only when doing so would be advantageous[9].

A detailed discussion of techniques for the mitigation of path explosion is beyond the scope of this report. The point that may be of interest to users of the Veri-J plug-in in relation to path explosion is that increases in program size and branching will, in principle, increase the plug-in's response time. However, this is unlikely to be noticeable with the fairly short code examples used in tutorials and assignments on the Reasoning sub-module.

2.2 SAT solvers

In principle, the satisfiability of a propositional logical formula (the 'SAT problem') could be determined by evaluating the formula for every possible combination of true or false values of its variables. However, each additional variable doubles the size of the solution space, making this brute force approach infeasible for all but trivial formulae[10].

The SAT problem is NP-complete. This means that although it is computationally intractable other than by brute force, it is easy to check whether any proposed solution is correct. SAT solvers use the second property to search the solution space heuristically, by trial and improvement.

The solver has many enhancements but its basic principle is similar to how a human would solve a Sudoku puzzle. It builds the solution incrementally, trying out one variable value at a time, checking each time if a solution remains possible and, if not, backtracking to the previous possible solution point and trying out another variable instead, deducing and propagating information as it proceeds.

The algorithm is complete – no false negatives - because it will return SAT if a solution exists. And it is sound – no false positives - because it will not return SAT if a solution does not exist.

Before considering in more detail how the algorithm works, it is important to note that the input to it must be in 'conjunctive normal form' or CNF, which is described next.

2.2.1 Conjunctive Normal Form

The logical formulas input to a SAT solver must be in 'conjunctive normal form' or CNF. This means that the whole formula is conjunction of clauses, each of which is a disjunction of literals. A literal may be either a variable or the negation of a variable. A variable may be present in any number of clauses. For example:

- $(A \vee B) \wedge (B \vee C)$
- $(A \vee \neg B \vee \neg C) \wedge (D \vee E \vee F)$
- $(A \vee B) \wedge C$
- $D \vee E$
- $E \wedge F$

A formula in CNF will be satisfiable if and only if there is at least one combination of its literal values that makes all of its clauses true. Formulae can be converted to CNF using De Morgan's and Distributive Laws and by cancelling out double negation:

Original Formula	CNF
$\neg\neg A$	A
$A \implies B$	$\neg A \vee B$
$\neg(A \wedge B)$	$\neg A \vee \neg B$
$\neg(\neg A \vee \neg B)$	$A \wedge B$
$(A \wedge B) \vee C$	$(A \vee C) \wedge (B \vee C)$

Next, we next explore how the SAT solver algorithm works, starting with a naïve implementation of backtracking and building on this improvement by improvement to explain the most recent developments[11][12][13].

2.2.2 Simple backtracking

A very basic SAT backtracking algorithm would be:

1. Choose any variable without an assigned value (or return SAT if all variables have been assigned)
2. Guess a truth value of true or false for the variable.
3. Check if all clauses in the formula remain potentially satisfiable.
    - If yes, return to Step 1 and chose another variable.
    - If no, has the variable been tried with both truth values (true and false)?
        - If no, return to Step 2 and assign the other truth value to the same variable
        - If yes, backtrack – i.e., undo all value assignments since the last time when all clauses in the formula were potentially satisfiable and continue.
        - Return UNSAT if there is nowhere to backtrack.

This algorithm works but is very slow because it makes a number of redundant checks. These are removed in the improved backtracking algorithm.

2.2.3 Improved backtracking

To describe the improvement, we refer to a 'positive' literal as one that evaluates to true and to a 'negative' literal as one that evaluates to false. As noted above, a literal is either a variable or the negation of a variable. So the negation of a true variable creates a negative literal and of a false variable creates a positive literal. And by itself, a true variable is a positive literal and a false variable is a negative literal.

The key insight is that once a variable has been assigned a value that leaves the formula potentially satisfiable, it is no longer relevant to the proof (unless the algorithm subsequently has to backtrack). Specifically:

- Any clauses in which the variable's literals are positive will evaluate to true and remain so.
- Any of the variable's literals that are negative will have no further effect on the clauses that contain them.

These true clauses and negative variables can be removed from further consideration (until and unless they need to be restored by a backtrack).

Backtracking is indicated when deleting a negative literal empties a clause completely, because the empty clause and thus the entire solution becomes unsatisfiable under the current assignment.

So with these improvements, the algorithm becomes:

1. Choose any variable without an assigned value (or return SAT if all variables have been assigned)
2. Guess a truth value of true or false for the variable.
3. Remove all clauses in which the variable's literals are now positive.
4. Remove all of the variable's negative literals.
5. Check if any empty clauses have been created

- If no, go to Step 1.
- If yes, has the variable been tried with both truth values (true and false)?
  - If no, return to Step 2 and assign the other truth value to the same variable.
  - If yes, backtrack – i.e., undo all assignments and deletions since the last time when all clauses in the were potentially satisfiable and continue. Return UNSAT if there is nowhere to backtrack.

2.2.4 DPLL algorithm

The DPLL or Davis-Putnam-Logemann-Loveland algorithm is named after the researchers who proposed it in 1962[11][12]. It introduces the improvement of unit propagation to the improved backtracking algorithm. If deletion of negative literals results in a clause containing only a single literal (a 'unit clause'), the variable concerned must be assigned whichever truth value makes that literal positive. This forced assignment may in turn create new unit clauses and so on. This is 'unit propagation', also known as Boolean Constraint Propagation (BCP).

With this improvement, the algorithm becomes:

1. Choose any variable without an assigned value (or return SAT if all variables have been assigned)
2. Guess a truth value of true or false for the variable.
3. Remove all clauses in which the variable's literals are now positive.
4. Remove all of the variable's negative literals.
5. Perform unit propagation if any unit clauses have been created.
6. Check if any empty clauses ('conflicts') have been created
   - If no, go to Step 1.
   - If yes, has the variable been tried with both truth values (true and false)?
     - If no, return to Step 2 and assign the other truth value to the same variable
     - If yes, backtrack – i.e., undo all assignments and deletions since the last time when all clauses in the were potentially satisfiable and continue. Return UNSAT if there is nowhere to backtrack.

Unit propagation or BCP can be illustrated by an example taken from a module on SAT solvers at the University of Washington[14]. It presents a logical formula in CNF for variables x_1 to x_7, contained in clauses C_1 to C_8 as illustrated in Figure 2.1.

Applying the DPLL algorithm to this formula:

1. Guess $x_1 == true$. This has no further effect
2. Guess $x_2 == true$. This makes C_5 a unit clause, requiring x_5 to be assigned.
3. Set $x_5 == true$. This makes C_2 and C_3 unit, requiring x_6 and x_7 respectively to be assigned.
4. Set $x_6 == true$ and $x_7 == true$. This causes a conflict by turning all literals in C_4 negative.
5. Backtrack to x_2, set $x_2 == false$ and repeat.

The formula is satisfiable and Figure 2.2 shows a decision tree for the guesses that DPLL might make to determine this. We can see in this case that the algorithm backtracked four times, making five attempts or 'runs' before it found a solution. In fact, it encountered the same conflict (i.e., at C_4) each time.

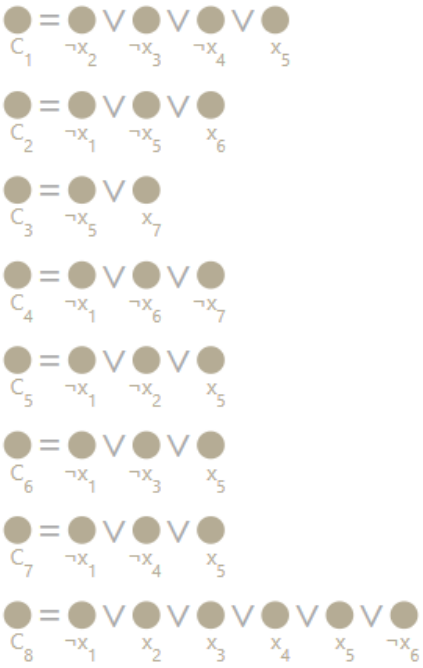


Figure 2.1: A logical formula in CNF[14]

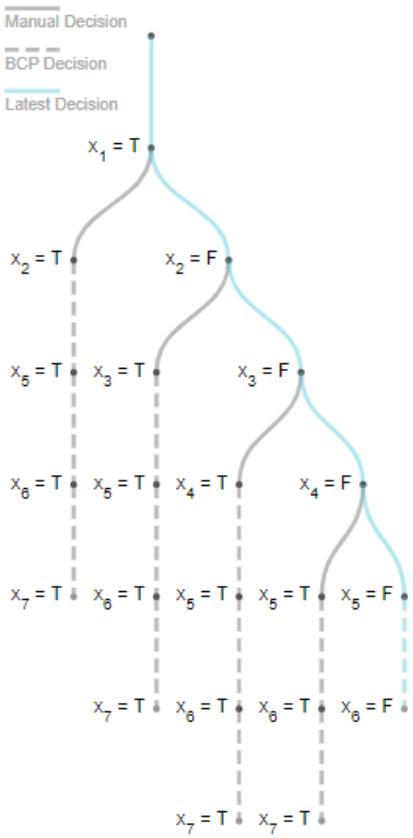


Figure 2.2: DPLL decision tree[14]

2.2.5 Conflict-driven clause learning

DPLL's main shortcoming is that it learns nothing from unsatisfiable assignments other than that they are unsatisfiable. It does not analyse an unsuccessful assignment to understand why it has caused a conflict. So, it is doomed to keep making the same mistake until it finds a solution or returns unsat.

Conflict-driven clause learning is an improvement to DPLL that addresses this weakness. Its core idea is to identify the combination of assignments that led to the conflict and not attempt that combination again. In the example above, the decisions (i.e., guesses) that led to the conflict in C_4 were $x_1 == true$ and $x_2 == true$. Each of these by itself may be part of a solution. However, the emergence of the conflict tells us that any search made after they have been made together is futile. In other words, $(x_1 \wedge x_2)$ represents an unsat region of the solution space. The search should be confined to its complement $\neg(x_1 \wedge x_2)$ or $(\neg x_1 \vee \neg x_2)$, known as a conflict or learnt clause.

This clause has two properties that make it useful:

- It is implied by the formula being evaluated. (We know that $\neg clause \wedge formula \implies false$, from which $formula \implies clause$).
- It evaluates to false under the conflict-causing partial assignment.

The first property means that the learnt clause can be added to the formula without changing the set of satisfying assignments. The second guarantees that after it has been added, the conflicting assignment will never be tried again. CDCL works by identifying a conflict clause after each conflict and adding it to the formula before the next run.

However, simply using all the guesses to make a conflict clause and flipping one of them would not be an efficient way to avoid the conflict. Instead, CDCL looks for an assignment that has been forced by the current guess and is closer to the conflict. We shall see how it does this next.

Implication graph

The CDCL algorithm keeps a record of the guessed and implied (i.e., BCP'd) assignments it makes in a directed acyclic graph known as an 'implication graph'. Each assignment is represented by a node, with a special 'K' node for the conflict. The edges point from one assignment to the others implied by it through BCP. Edges are labelled with the name of the unit clause in which the implied assignment took place. Figure 2.3 shows the implication graph for the assignments that lead to the conflict in the example above. In addition to the assignment, each node also displays the decision level of the guess (e.g., '@1') at which the assignment was made or implied.

Each of the four cuts superimposed on the graph in Figure 2.3 separates the guessed assignments from the conflict. We can see that, at each cut, if any implied (i.e., BCP'd) assignments in the starting nodes of the cut edges had instead been guessed, the same conflict would have arisen, via the same implied assignments as below the cut. This means that we can define a conflict clause for any cut by treating any implied nodes immediately above it as guesses. In this way, conflict clauses at Cuts 1 to 4 can be determined as $(\neg x_1 \vee \neg x_2)$, $(\neg x_1 \vee \neg x_5)$, $(\neg x_1 \vee \neg x_6 \vee \neg x_5)$ and $(\neg x_1 \vee \neg x_6 \vee \neg x_7)$ respectively. However, not all of these are useful because the conflict clause needs to contain exactly one variable assigned at the current decision level. This variable then turns unit within the conflict clause during the next run of the algorithm, which avoids the conflict-causing assignment.

To identify the useful clauses, we define a 'unique implication point' or UIP as being any node in the implication graph, apart from the conflict, that is on all paths from the most recent guess to the conflict. In the implication graph above, the most recent guess is $(x_2 = T@2)$. The two UIPs are $(x_2 = T@2)$ and $(x_5 = T@2)$. The first and last UIPs are the ones closest and furthest away from the conflict respectively.

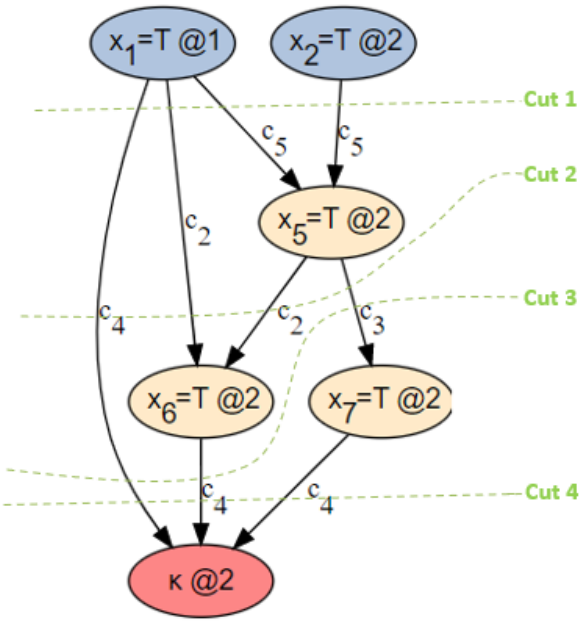


Figure 2.3: CDCL implication graph[14]

Any clause on a cut immediately below a UIP can be used as a conflict clause. However, the choice involves a compromise. Clauses close to the conflict tend to be smaller and therefore more efficient as they prune more of the sample space. However, clauses further way, being larger, contain more information about the conflict, which is useful later if the algorithm needs to backtrack and explore a completely different subtree. The 'First-UIP' choice appears to be more popular[15].

After adding the chosen clause to the formula, the algorithm backtracks to the most recent guess, other than the current one[3], that affects a literal in the clause[4]. The clause will then become unit. The unit literal is the negation of the UIP and its forced assignment avoids the conflict.

Figure 2.4 shows the use of CDCL with $C_9 = (\neg x_1 \vee \neg x_5)$ as the conflict clause. The first run ends with the conflict as before. Then the algorithm adds the new clause and backtracks to $x_1 = true$, which makes C_9 unit. In this simple example, unit propagation is then able to find a solution without any further guesses. We can see that CDCL has arrived at a result in just two runs compared to five for DPLL.

As conflict clauses define unsat regions of the solution space, the solver stores them for reuse if, as is likely, it needs to backtrack and explore another subtree. Also, a conflict clause may imply a reason for the conflict, so can be useful afterwards to help understand why an unsatisfiable problem is not satisfiable.

However, unrestricted addition of learnt clauses to the formula can cause problems as it increases the time needed for unit propagation. Also, the conflict clauses themselves can become very long as the search progresses, adding to the problem. CDCL solvers use size-bounded and/or relevance-bounded learning strategies to manage this problem, at the expense of losing some of the information in the learnt clauses. In size-bounded learning, the solver does not store clauses above some arbitrary size. In relevance-bounded learning, it periodically drops learnt clauses in which the number of unassigned variables is greater than some arbitrary limit[16].

[3]The current guess triggered the conflict, so is going to be replaced by the UIP variable.
[4]This is known as non-chronological backtracking.

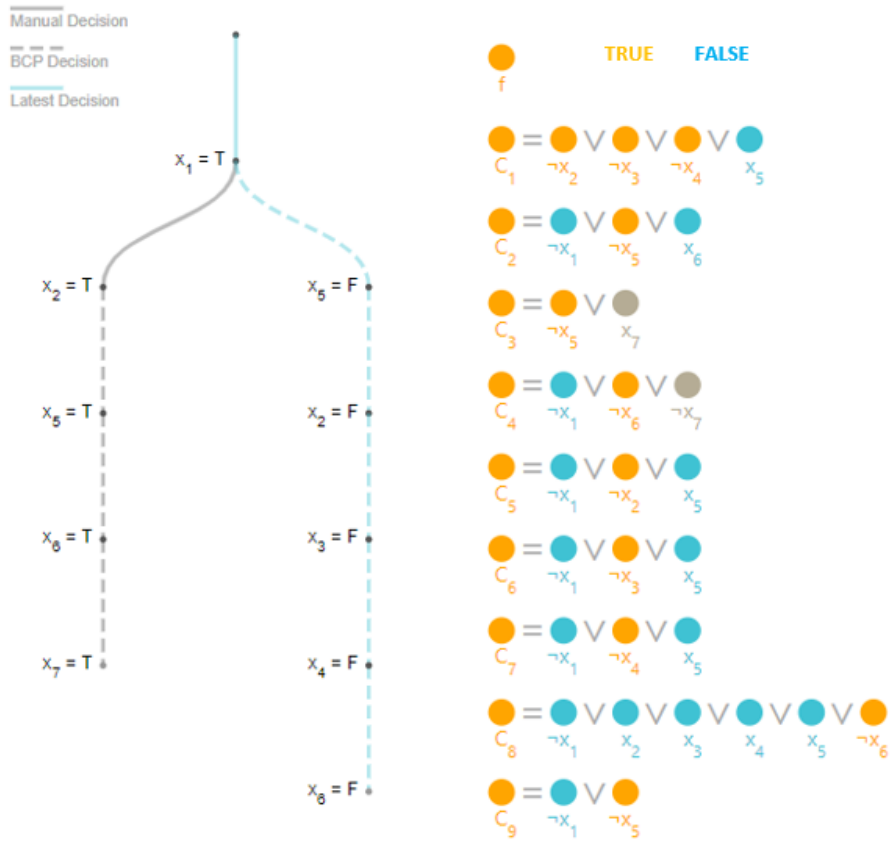


Figure 2.4: CDCL decision tree and solution[14]

2.2.6 The two watched literals technique

After each variable assignment, whether guessed or implied, has been propagated through the formula being searched, the algorithm has to check for new unit clauses. A naïve approach to this would be to visit every clause and count how many unassigned literals it contained. A better approach comes from noting that a clause cannot be unit if it has two or more unassigned literals.

Before the algorithm begins, it arbitrarily selects two literals from each clause to watch during the search. If either of these literals subsequently evaluates to true, both can be dropped from the watch because the underlying clause will no longer be in play as it will also be true. However, if either literal evaluates to false, it is dropped from the watch and the underlying clause is searched for another unassigned literal to watch instead. If no such literal can be found then the clause has become unit and the literal still in the watch is the unit literal. It is assigned a value and dropped. Dropped pairs are reinstated if the algorithm backtracks. . This approach ensures that the algorithm only visits clauses that have the potential to become unit clauses. It is known as the 'two watched literals' technique[15].

2.2.7 Decision heuristics

In both DPLL and CDCL, the choice of variable for the next decision (i.e., guessed assignment) has a considerable effect on performance. Generally, it is better to try out literals that occur in numerous clauses before ones that occur in only a few as this may identify conflicts and/or a solution more quickly. Some techniques for this are:

- Dynamic Largest Individual Sum (DLIS). This simply involves choosing the literal that satisfies the most unresolved clauses.[17]

- Pure literal elimination. A pure literal is one that has the same polarity (i.e., either negated or not negated) in all clauses in which it occurs. Pure literal elimination involves identifying such variables and assigning them whatever value makes the literals positive, so that the containing clauses can be eliminated from the search under the current partial assignment[15].
- Variable State Independent Decaying Sum (VSIDS). A list of all literals is created up-front, sorted in descending order of their frequency of occurrence across all clauses. Whenever the solver needs to make a guess, it takes the next literal off the list. Variables involved in more recent conflicts are moved up the list, to be selected sooner[17].

Both DLIS and pure literal elimination have the benefit of being straightforward and intuitive. However, both are expensive as the complexity of finding the literal is proportional to the number of unresolved clauses. More importantly, both defeat the purpose of the two watched literal technique, which is to avoid the overhead of checking the contents of each clause. VSIDS is generally considered superior to both.[15][16]

2.2.8 CDCL's potential in program verification

Z3, the SAT solver used by Dafny for program verification is based on the DPLL algorithm. However, there is considerable evidence that CDCL significantly outperforms DPLL[18]. A CDCL-based software verifier might achieve a considerable improvement over DPLL both in speed and in the size and range of problems that can be solved. Furthermore, it might not need the user to provide as many lemmas and loop variants, making writing proofs closer to human intuition.

Another interesting possibility arises from the potential of CDCL conflict clauses to provide information for optimizing programs that verify and counterexamples for those that do not. As we saw in section 2.1, solvers attempt verification by trying to solve the negation of a formula. So if a program verified, the clauses would show why it had verified (i.e., why its negation could not be solved). Very long clauses, indicating areas of complexity that the program relied on for correctness, might provide unexpected insights into how the code could be optimized. Similarly, for proofs that fail (i.e., the negation could be solved), conflict clauses contain information about why the proof might have succeeded. Therefore, their negation might imply useful counterexamples.

2.3 SMT solvers

The SMT (Satisfiability Modulo Theories) solver can be thought of as a two-way interface between the SAT solver and the real world. Real-world problems seldom present themselves as Boolean formulae. Rather, they are expressed using domain specific theories such as linear equalities and inequalities, arrays, polynomials and so on[19].

The input to an SMT or theory solver is a problem modelled in a relevant theory. The SMT solver abstracts this model into a Boolean formula by mapping the atoms of the model into Boolean variables. The abstracted formula is then processed by the SAT solver. Any solution produced by the SAT is returned to the theory solver, which checks it for validity in the relevant theory. If the solution passes this test, it is output as a solution by the theory solver.

However, if the SAT solution is not theory-valid, the theory solver identifies the part of the solution that is in conflict and abstracts this into a Boolean clause known a 'theory lemma'. The theory lemma is added as an additional constraint to the original formula, which is then resubmitted to the SAT solver. These steps are repeated until a theory-valid solution is found or the SAT solver returns unsat. The process always converges because there are a finite number of atoms in the input model and therefore a finite number of theory lemmas that can be created by the theory solver.

The 'modulo' in 'SMT' refers to the theory solver's role of modulating solutions returned by the SAT solver to accept only those that are theory-valid.

The process can be illustrated by an example taken from the official online Z3 tutorial[20]:

The linear model:

$$(x >= 0) \wedge (y == x + 1) \wedge ((y > 2) \vee (y < 1))$$

can be abstracted to the Boolean propositional formula:

$$p1 \wedge p2 \wedge (p3 \vee p4).$$

This formula is satisfiable and a possible solution is:

$$p1 == true, p2 == true, p3 == false, p4 == true$$

which de-abstracts to:

$$(x >= 0) \wedge (y == x + 1) \wedge (\neg(y > 2)) \wedge (y < 1).$$

This is not theory-valid because its subset

$$(x >= 0) \wedge (y == x + 1) \wedge (y < 1)$$

is not theory valid. To avoid this assignment, the theory solver adds the following blocking clause (theory lemma) to the original formula:

$$\neg p_1 \vee \neg p_2 \vee \neg p_4$$

The SAT solver then produces a new truth assignment

$$p_1 = true, p_2 = true, p_3 == true, p_4 = false$$

which de-abstracts to the theory-valid solution:

$$(x >= 0) \wedge (y == x + 1) \wedge (y > 2) \wedge \neg(y < 1)$$

Any unsatisfiable subset of a set of theory literals is known as a 'justification' of its parent set. A justification is non-redundant if it has no justifications other than itself. It is highly desirable that justifications (theory lemmas) created by the theory solver should be non-redundant as this can significantly reduce the search space. For example, if the theory solver needs to tell the SAT solver that if some variable 'x' is less than three it cannot also be greater than five, it would be redundant to further add that x cannot be greater than, say, eight. The redundant variable will potentially increase the number of searches that the SAT solver performs without affecting the outcome.

Although conceptually distinct, in practice the SMT and SAT solvers are closely integrated and work together in lockstep. Typically:

- Rather than waiting for complete solutions to be output from the SAT solver, the theory solver checks partial truth assignments for unsatisfiability as they are being explored by the SAT solver.
- Theory deduction rules are used to propagate truth values during the SAT search, which reduces the space that the SAT solver needs to explore. For example, if the SAT solver assigns a value of true to, say, x $\geq$ 18 then the theory solver will tell it that, say, x $\geq$ 5 must also be true.
- SMT solvers can backtrack with the SAT solver. Theory lemmas added while one partial truth assignment is being explored are removed if not relevant to another potential solution.

The intertwining of DPLL-based Boolean reasoning with SMT-based deduction for a theory T is referred to as DPLL(T).

Z3 is a DPLL(T) solver but, confusingly, the online tutorial[20]describes it as a CDCL(T) solver and offers the example taken above as a demonstration of its CDCL capability. This is incorrect as the blocking clause added in that example is a theory lemma, not a CDCL conflict clause. It may be useful to note the difference between the two:

- A CDCL conflict clause is derived internally by the SAT algorithm, using only Boolean logic to search for a satisfying combiination of truth value assignments. By contrast, a theory lemma is derived externally to the SAT algorithm by the theory solver, by deduction within the theory concerned.

- A CDCL conflict clause is a pseudo-constraint that does not change the set of satisfying assignments. It is merely a reminder from the SAT algorithm to itself not to re-enter a region of the solution space where there are no solutions. By contrast, a theory lemma is a genuine additional constraint that excludes combinations of truth value assignments that would be returned as solutions without it.

2.4 Z3 theorem prover

Z3, the theorem prover used by Dafny for program verification, was originally developed at Microsoft Research by Nikolaj Bjørner and Leonardo de Moura, who began work on it in 2006. The first stable release was issued in 2012. It was open-sourced in 2015 under the MIT License and the source code is now hosted on GitHub[21].

Default input format to Z3 is through a command line interface that utilizes the standardized logical formula input formal known as SMT-LIB. Z3 also has official bindings for Python, C#, C, C++, Java and OCaml and some other languages as well.

Z3 has built-in support for over thirty theories. Some of the most commonly used are linear equations, linear inequalities, polynomial equalities, arrays, algebraic data types, bit-vectors and uninterpreted functions. Although Z3 does support polynomial equalities, in general it has limited support for nonlinear equations. It does not support problems that involve exponentials, logarithms or trigonometric functions. The DML&R Reasoning sub-module syllabus covers only linear equations and inequalities and arrays, all of which are supported by Z3 and Dafny.

Z3 is based on the DPLL algorithm, which is over sixty years old. As described in section 2.2.8, a move to CDCL could deliver significant benefits. Z3 is an ongoing project and there are already hints on its website that this is being considered[21].

In the meantime, despite its being based on a technology that many consider to be outdated, Z3 is in no danger of falling into disuse. It has a large user base because it was perhaps the first SMT/SAT solver to move out of the research laboratory into the wider software development community. Another reason for its longevity is that it is used by the software proofing language Dafny as its verification engine. Dafny is one of the leading tools for this purpose, and we explore some of its features next.

2.5 Dafny: an overview

Dafny is an imperative programming language with built-in support for inline program specification constructs. It was developed at Microsoft Research and first released in 2009. The source code is now hosted on GitHub under the open-source MIT license[22]. Dafny uses the Z3 SMT-SAT solver as its verification engine. It interfaces with Z3 via a symbolic interpreter known as Boogie, also developed at Microsoft Research[23] and open-sourced on GitHub[24]. Boogie returns proof results from Z3 to Dafny, along with counter-examples if relevant.

Verification in Dafny checks for total correctness – that is, not only that the programs implements the specification but also that it terminates. Its architects also claim it to be sound; in other words, it will not 'prove' an incorrect program to be correct[25]. Like the technique taught on the Reasoning sub-module, specifications and verification in Dafny are modular. A proof does not need to be repeated when the program concerned is included as a component in a larger program[25].

2.5.1 Imperative features

Dafny implements the basic features of an imperative programming language including field and variable declarations and assignments, subroutines, branching, loops and arrays. It also supports reference type declarations but generally has limited support for object-oriented programming. The essential arithmetic and logical operators are implemented. Exceptions include exponention, increment/decrement and bitwise operators.

The language offers four basic data types: 'bool' for Booleans, 'int' for integers, 'real' for real numbers and 'char' for characters. In addition to arrays, three collection types are supported: sets, multisets, and sequences. These have the properties that the type names suggest and support relevant operations between variables of each type and to the elements within them. They are used more in specifications than in programming code, to describe the existing or required properties of variables[26].

A useful feature of subroutines ('methods') in Dafny is their ability to have more than one return variable. Being able to refer to multiple return variables can make method postconditions shorter and clearer.

2.5.2 Specification features

Program specification in Dafny is based primarily on the use of method pre and postconditions, inline propositions (mid conditions) and loop invariants. Dafny has constructs to express all of these effectively and also for loop variants.

Some of Dafny's specification features are described below. The list is not comprehensive. Rather, it includes just a few key components, chosen to convey a flavour of the language's proofing capabilities.

Functions

Dafny offers a special type of subroutine known as a 'function', solely for use in specification annotations. A function cannot have side-effects or be assigned. It consists solely of one expression and has a single, unnamed return value. The following function, taken from the Dafny introduction[27], computes the value at position 'n' in the Fibonacci sequence:

```
function fib(n: int): int
requires n >= 0;
{
    if n == 0 then 0 else
    if n == 1 then 1 else fib(n 1) + fib(n 2)
}
```

The ability to use recursion in functions makes them particularly useful in loop invariants, as shown in the Dafny method below for computing Fibonacci numbers:

```
method ComputeFib(n: int) returns (b: int)

requires n >= 0;      //method precondition
ensures b == fib(n); //method postcondition

{
    if (n == 0) { return 0; }

    var i := 1;
    var a := 0;
    b := 1;
```

```
    while (i < n)
    invariant 0 < i <= n;  //loop invariants
    invariant a == fib(i 1);
    invariant b == fib(i);
    {
        a, b := b, a + b;  //parallel assignment
        i := i + 1;
    }
}
```

Quantifiers

Dafny supports the use of the logical quantifiers 'forall' and 'exists' and logical connective operators for implication, equivalence and reverse implication (follows from) in its annotations. The example below, also taken from the Dafny introduction[27], illustrates these in a method that returns the index of an input value in an integer array or -1 if the value is not found:

```
method Find(a: array<int>, key: int) returns (index: int)
requires a != null;
ensures 0 <= index ==> index < a.Length && a[index] == key;
ensures index < 0 ==> forall k
    :: 0 <= k < a.Length ==> a[k] != key;
{
    index := 0;
    while (index < a.Length)
    invariant 0 <= index <= a.Length;
    invariant forall k :: 0 <= k < index ==> a[k] != key;
    {
        if (a[index] == key) { return; }
        index := index + 1;
    }
    index := -1;
}
```

The method's postconditions use the forall quantifier and the implies connective to verify firstly, that if the returned index is zero or greater, the input value does in fact exist at the returned index and secondly, that the input value has in fact not been found if the method returns -1. The loop invariant uses the forall quantifier and the implies connective to confirm that the value sought has not already been found before continuing with the next iteration.

Lemmas

Dafny supports the use of lemmas or ghost methods, which are theorems used to prove another result. Their usefulness can be seen in an example taken from Dafny's GitHub repo[22]. The problem is to prove the correctness of the code that finds a zero value in an integer array a that has two properties:

- all values in the array are whole numbers
- each of its elements is either equal to, or one less than, the element before it.

These properties allow the code searching for zero to skip values in the array. For example, if `a[j] == 5`, then a zero cannot occur until position `j + 5` at the soonest. In general, the code can skip from any position `j` to position `j + a[j]` without having to check anything in between. The Dafny implementation of this is:

```
method FindZero(a: array<int>) returns (index: int)
requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
ensures index < 0  ==> forall i
    :: 0 <= i < a.Length ==> a[i] != 0
ensures 0 <= index ==> index < a.Length && a[index] == 0
{
   index := 0;
   while index < a.Length
      invariant 0 <= index
      invariant forall k :: 0 <= k < index && k < a.Length ==> a
          [k] != 0
   {
      if a[index] == 0 { return; }
      index := index + a[index];
   }
   index := -1;
}
```

The code loops through the array, skipping the positions it does not need to check. It returns the loop counter's value if it finds zero or -1 if the loop finishes without its doing so. The method's preconditions ('requires' statements) establish that the array has the two properties described above. The two postconditions ('ensures' statements) establish respectively that the method is sound and complete. Before and after each iteration, the invariants check that the index is zero or more and that the array element at the loop's current position is not zero.

The method executes correctly but Dafny will not verify it. Instead, it returns an error message claiming that the loop will not maintain the second invariant. This is because Dafny knows only about the relationship between successive elements in the array (from the second precondition). Unlike the human mind, it cannot extrapolate from this to understand how elements further apart are related. The invariant applies to every element in the array but the loop does not check them all. So Dafny rejects the invariant because it cannot be sure that the skipped elements are not zero. It needs to be told this explicitly with a lemma.

Dafny lemmas are implemented as void methods whose pre- and post-conditions express the theorem needed. Most often, the theorem establishes a relationship between the method's input parameters. The lemma for the example above needs to establish that none of the elements in array `a` from location `index` to location `index + a[index]` can be zero. This becomes its postcondition. The variables need by this postcondition, `index` and `a`, are provided as the lemma's input parameters:

```
lemma SkippingLemma(a: array<int>, j: int)
requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
requires 0 <= j < a.Length
ensures forall i :: j <= i < j + a[j]
    && i < a.Length ==> a[i] != 0
```

Like the method itself, the lemma needs the preconditions that establish the two properties of the array. Because `index + a[index]` may equal or exceed the array's length, the third precondition is needed to keep `j` within range.

The lemma is applied by calling it from the place in the code where its theorem needs to be asserted:

```
method FindZero(a: array<int>) returns (index: int)
requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
ensures index < 0  ==> forall i
   :: 0 <= i < a.Length ==> a[i] != 0
ensures 0 <= index ==> index < a.Length && a[index] == 0
{
   index := 0;
   while index < a.Length
      invariant 0 <= index
      invariant forall k :: 0 <= k < index && k < a.Length ==> a
          [k] != 0
   {
      if a[index] == 0 { return; }
   SkippingLemma(a, index);      //lemma assertion
      index := index + a[index];
   }
   index := -1;
}
```

Using the lemma, Dafny can now verify that the method is correct. It does not require that the lemma be proved first. This is useful because it allows to the programmer to check that the lemma works before making the effort to prove it. The proof is optional and is implemented as the lemma's code block. The `SkippingLemma` proof has been omitted from this report for brevity but is available from the source referenced[28].

Other useful resources on Dafny include an introductory tutorial[27] and video[29] by founding architect Ruston Leino, and an introduction to the use of lemmas[30]. A language grammar annotated with detailed notes provides the authoritative reference about its capabilities[31].

Chapter 3

Design

This chapter describes the features and capabilities of Veri-J. It focuses on the plugin's functionality, with reference to the needs of the DML&R Reasoning sub-module.

3.1 User interface

Veri-J's user interface is contained within a single tool window. This appears automatically during start-up, at the bottom of the IDE's main window. It has a 'lightening' icon on its tab, as shown in Figure 3.1 below.

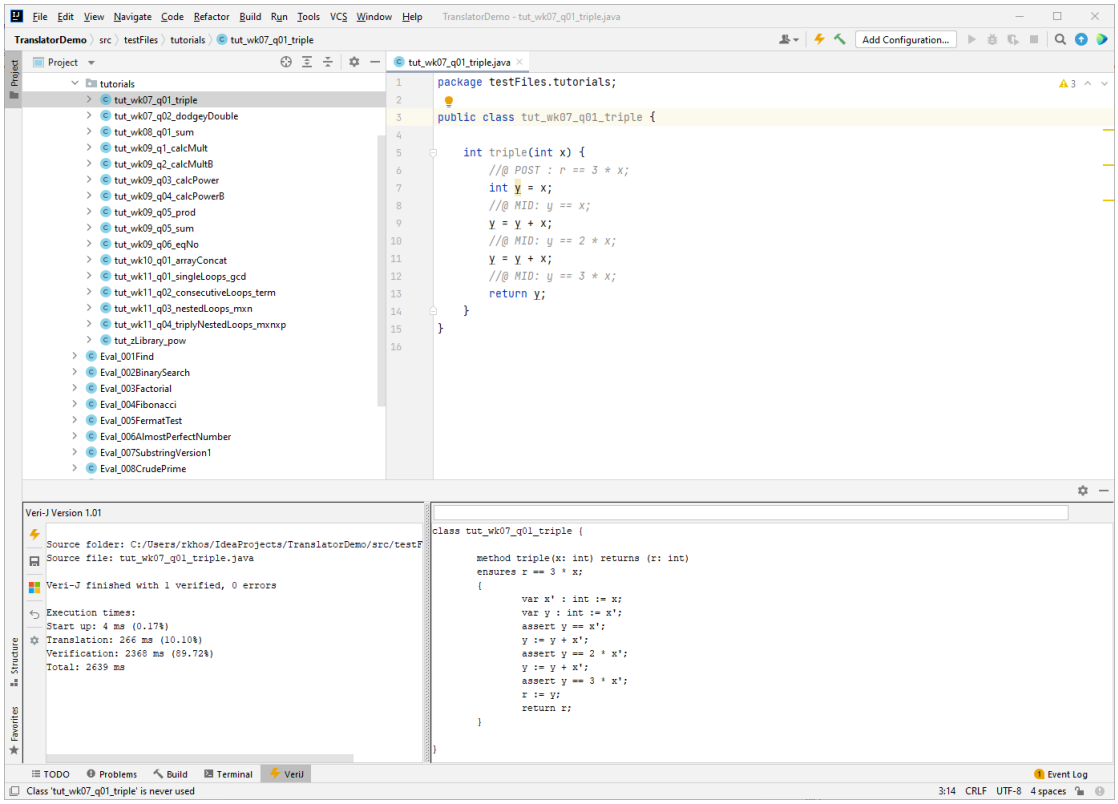


Figure 3.1: Veri-J Tool Window

The tool window is split vertically into a feedback display pane (on the left) and a Dafny display pane (on the right). The vertical toolbar along the tool window's left edge provides access to the

plugin's functionality.

The plugin operates on the file open in IntelliJ's current editor pane, that is, the last editor pane to contain the caret. The toolbar has five actions:

Verify checks the file for syntax and semantic errors and then attempts to verify it. Feedback appears in the pane to the left. The Dafny code generated appears in the pane to the right.

Save opens a file saver dialog to save the Dafny code generated by Veri-J.

VSC opens the saved Dafny file in Visual Studio Code.

Reset clears the feedback and Dafny panes along with any error highlighting in the Java input file.

Settings opens the Settings dialog box (Used during installation to save the locations of the Dafny and VSC executables).

3.2 Error handling

When the Verify button is clicked, Veri-J checks for errors at three levels: syntax, semantic and verification. Errors are checked and reported one level at a time only. The plugin will not attempt translation if it detects syntax errors. And it will not attempt verification if it detects semantic errors during translation.

The translator checks for some semantic errors as it converts the Java code to Dafny. For example, it prevents the use of Dafny keywords as identifiers in the input code. It also prevents the use of some of the Java constructs that are not included in the translation because they are not relevant to the DML&R Reasoning sub-module.

Verification errors are detected and fed back to Veri-J from the Z3 SMT/SAT solver.

Errors of any sort are underlined and highlighted in red in the Java input file. Error coordinates (line number and position) and description are displayed in the feedback pane.

Syntax errors are checked interactively as well as being trapped by the Verify button. Error highlighting and messages will appear and disappear as code is typed in the current editor.

3.3 Writing Java code for Veri-J

Veri-J supports all the features of Java needed on the Reasoning sub-module:

- Class, field, method and local variable declarations.
- Method calls.
- Branching with if/else (but not switch/case statements).
- While loops (but not do/while or for loops).

- Return statements.
- Break statements.
- Statement blocks, with or without an identifier.
- Most Java expressions. (Not included are lambda expressions and expressions relevant only to OO programming such as method references and type comparisons).
- All eight Java primitive data types and strings.
- Single- or multi-dimensional arrays of any of the primitive data types.

Dafny does not support the unary increment/decrement expressions. It also restricts use of method calls. For reasons explained in section 4.5.9, these restrictions have also had to be applied to the use of Java in Veri-J:

- The increment/decrement expression, whether pre- or post-fix, can be used only as a standalone statement, not contained within other expressions. So, it is valid to write:

```
i++; //standalone statement
```

  But not:

```
myVar1 = i++;
myVar2 = myArray[--k];
```

- Method calls must be either standalone statements or comprise the entire right-hand side of an assignment. For example, it is valid to write:

```
//standalone statement:
myMethod(a, b);
//entire right-hand side of an assignment:
int[] myVar1 = myMethod(a, b);
// entire right-hand side of an assignment:
myVar2 = myMethod(a, b);
```

  But these are invalid:

```
//method call within expression:
myVar2 = myMethod(a, b) * 3;
//method call within expression:
myVar2 = myMethod(myMethod2(), b);
```

In both cases, the workaround is to break up the compound expression by assigning its parts to intermediate variables.

3.4 Writing verification statements in Veri-J

3.4.1 Basic syntax

The basic structure common to all the verification statements (or annotations) in Veri-J is:

```
//@ keyword: expression;
```

Like Java statements, verification statements may span or share lines. Whitespace between their components is ignored.

As the list below shows, the syntax of these statements in Veri-J is similar to the notation used on the Reasoning sub-module:

Statement type		*keyWord*		expression type	
Pre-condition	//@	PRE	:	Boolean	;
Post-condition	//@	POST	:	Boolean	;
Mid-condition	//@	MID	:	Boolean	;
Invariant	//@	INV	:	Boolean	;
Variant	//@	VAR	:	Integer	;

In addition to the five main proof statements above, Dafny (and Veri-J) require two further statements to control access to reference-type variables such as arrays:

Modifies	//@	MODIFIES	:	Reference	;
Reads	//@	READS	:	Reference	;

Use of the Modifies and Reads statements is described in sections 3.4.3 and 3.5.5. respectively

All five elements are required in all statements, from the starting '//@' tag to the trailing semicolon. The keyword must be in uppercase. The '@' character in the starting tag is important as without it Veri-J's parser matches the line to a comment and ignores it. All other syntax errors are detected and reported.

Regardless of its type, the return parameter (if any) of an executable method must be referred to as r in method postconditions but nowhere else. It can be given any other valid name if required in the method body. Use of the identifier `r` for any purpose outside a postcondition will cause an error.

3.4.2 Location

Annotation statements in Veri-J have the same locations as they have on paper in the Reasoning sub-module:

- Method pre- and post-conditions are placed after the method declaration but before the opening curly bracket of the method block. An unlimited number of either type of condition may be used, in any order.
- Loops may have an unlimited number of invariant statements and, if needed, no more than a single variant statement. The variant must be placed after the invariants. These statements are placed immediately before the loop declaration (i.e., the while statement). There must not be any other code between them and the loop declaration.
- Mid-conditions can be placed wherever they are needed in a method or loop code block.

The example below (Week 9 tutorial, Q1) illustrates the basics of the notation[1].

[1] Veri-J solutions for all tutorial questions from Week 7 to Week 11 are available for download in a zip file along with several further examples

```
public class tut_wk09_q01 {

    int calcMult(int m, int n)
    //@ PRE: m >= 0 && n >= 0;
    //@ POST: r == m * n;
    {
    int cntr = 0;
    int acc = 0;

    //@ INV: acc == n * cntr && cntr <= m;
    //@ VAR: m - cntr;
    while (cntr < m) {
        acc = acc + n;
        cntr = cntr + 1;
        }

    //@ MID: acc == m * n;
    return acc;
    }
}
```

3.4.3 Modifies statement

Dafny (and Veri-J) require any heap memory being changed by an executable method to be explicitly declared[2]. This is achieved by inserting a Modifies statement after all other specification statements for the method declaration. The syntax of a Modifies statement in Veri-J is:

```
//@ MODIFIES: identifier (, identifier)*;
```

For example, the Modifies statement in Veri-J for a Java method that needed to modify three arrays a, b and c would be:

```
//@ MODIFIES: a, b, c;      //case-sensitive
```

3.4.4 Operators

Veri-J provides all the operators needed to construct Boolean expressions in verification statements. Mostly, these are the same as used in Java:

Comparison Operators:	
==	Equality
!=	Disequality
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

[2]The reasons for this relate to memory management by Dafny during the verification process and are outside the scope of this project.

Logical Operators:	
&&	Conjunction (logical AND)
||	Disjunction (logical OR)
!	Negation (complement)
^	Bitwise exclusive OR (XOR) [3]
==>	Implies
<==	Reverse implication (follows from)
<==>	Equivalence (if and only if)

Quatifiers	
forall	Universal quantifier
exists	Existential quantifier

Veri-J supports use of all Java arithmetic operators needed in verification statements.

3.4.5 Comparison chains

Veri-J supports chaining of the inequality operators as a shortcut to compare several expressions in one go. For example:

```
A <= B < C
```

is the same as:

```
A <= B && B < C
```

A chain can have an unlimited number of expressions. As in the example, a mix of strict and inclusive operators is allowed but they must be either all ascending or all descending. The example below illustrates the use of comparison chains, quantifiers and Boolean operators in the specification of a method that performs a binary search to find the index of a value in an array:

[3] Veri-J does not accept the bitwise AND ('&') and bitwise OR ('|') operators as these operators are not supported in Dafny.

```
int BinarySearch(int[] a, int key)
//@ PRE: a != null;
//@ PRE: forall j, k :: 0 <= j < k < a.length
         ==> a[j] <= a[k];
//@ POST: 0 <= r ==> r < a.length && a[r] == key;
//@ POST: r < 0 ==> forall k :: 0 <= k < a.length
          ==> a[k] != key;
{
    int low = 0, high = a.length, index;

    //@ INV: 0 <= low <= high <= a.length;
    //@ INV: forall i :: 0 <= i < a.length
        && !(low <= i < high) ==> a[i] != key;
    while (low < high)
    {
        int mid = (low + high) / 2;
        if (a[mid] < key) {
                low = mid + 1;
        } else if (key < a[mid]) {
            high = mid;
        } else {
            index = mid;
            return index;
        }
    }
    index = -1;
    return index;
}
```

3.4.6 Arrays

Veri-J supports the use of arrays[4] in verification statements. Arrays and individual array elements are referred to in the same way as in Java.

3.4.7 Array slicing

Array slices are supported as used on the Reasoning sub-module, with a slightly extended syntax to support both inclusive and excusive bounds at either end. The syntax of an array slice in Veri-J is:

identifier ('[' | '(') lowerBound? .. upperBound? (']' | ')')

where:

[4]The term 'array' is used throughout this report to mean one-ranked (one-dimensioned) arrays. Veri-J does accept and translate arrays of any rank to Dafny. However, multi-dimensional arrays are not currently used on the Reasoning sub-module.

Element	Description
identifier	array name
('[' | '(')	'[' is inclusive '(' is exclusive
lowerBound	Any valid integer expression Defaults to zero if omitted.
..	'..' Double dot range separator
upperBound	Any valid integer expression Defaults to array length if omitted
(']' | ')')	']' is inclusive ')' is exclusive

For example:

Slice	Includes
a[..)	All elements in array a.
a[i..j]	All elements from a[i] to a[j] inclusive
a[i..j)	All elements from a[i] to a[j-1] inclusive
a(i..j)	All elements from a[i+1] to a[j-1] inclusive
a[..]	Error - upper bound out of range.

3.4.8 Array concatenation

Any number and mix of arrays and array slices can be concatenated using the colon (':') operator in a manner identical to the notation used manually on the Reasoning sub-module. For example, the expression:

```
a[i..j] : b[k..l) : c[..)
```

represents an array or array slice comprised of `a[i]` to `a[j]` followed by `b[k]` to `b[l-1]` followed by the whole of array c.

3.4.9 Array comparisons

Arrays, array slices and array concatenations can be compared to each other using the '==' and '!=' operators. For example:

```
a[i..j] : b[k..l) : c[..) == d[5..17) : e[..)
```

The '<' (is a 'proper' slice of) and '<=' (is a slice of) operators are also supported. These operators can be chained. For example, the expression:

```
a[i..j] < b[k..l) <= c[..)
```

requires that the sequence of elements in slice `a[i..j]` must also occur somewhere in the slice `b[k..l)` without comprising all of it. `b[k..l)` must in turn occur somewhere in array `c` or comprise all of it.

3.4.10 Array aggregates

On the Reasoning sub-module, array aggregates are denoted by the Greek symbols Σ for aggregate sum and Π for aggregate product. In Veri-J, these are denoted by the (case-sensitive) keywords 'Sigma' and 'Pi'. They can be applied to whole arrays, array slices or array concatenations. For example, the Veri-J postcondition:

```
//@ POST: r == Pi a[0..a.length);
```

requires that the method's return value `r` be equal to the product of all elements of array `a`.

3.4.11 Entry-state values

The entry-state value of a method input parameter is its value on entry to the method. Method specifications frequently need to compare current-state values with entry-state values.

On the Reasoning sub-module, the entry-state value of an in-parameter is known as its 'pre' value and represented by the subscript 'pre' after the variable's name. The subscript may be applied to both primitive and array-type variables. For example: x_{pre}, $(a[3])_{pre}$ or $(b[i..j))_{pre}$.

In Veri-J the pre value of a method in-parameter is denoted by the case-sensitive function `Pre()`. So, the examples given would become `Pre(x)`, `Pre(a[3])`, and `Pre(b[i..j))`.

In keeping with module requirements, `Pre()` values of arrays, array elements and array slices can be concatenated. For three arrays `a`, `b` and `c`, the expression:

```
Pre(a[3]) : b[m..9] :  Pre(c)
```

represents an array or array slice that starts with the entry-state value of `a[3]` followed by the current values of `b[m]` to `b[9]` followed by the sequence of all values in the entry state of `c`.

3.5 Verification method-type constructs

3.5.1 Overview

Dafny (and therefore Veri-J) offers three method-type constructs to assist in proofs – functions[5], predicates and lemmas. Functions provide a way for the specification to check the value of variables at the start, during or end of code execution. Predicates are functions in a shortcut syntax that return a Boolean. Lemmas are mini proofs that are used to split proofs that are otherwise too complex to be verified into smaller, manageable stages.

Functions, predicates and lemmas cannot change the values of reference-type variables. This includes the values held in arrays.

[5]In most programming languages, including Java, the terms 'method' and 'function' are used interchangeably. However, in this report, use of 'function' is reserved for the specification construct being described here and 'method' refers to an executable method.

3.5.2 Functions

A Veri-J function is similar in concept to a mathematical function. It returns a single value and has no side-effects. Here is a function that calculates the nth Fibonacci number:

```
//@ FUNC           //line break required after FUNC
public static int fib(int n)
//@ PRE: n >= 0;
{
    return (n == 0)? 0 : ((n == 1)? 1 : (fib(n - 1)
        + fib(n - 2)));
}
```

Function syntax has the following features:

- It begins with the Veri-J tag followed by the FUNC keyword and a line break (required).
- The modifiers public and static are required.
- The syntax and location of any pre- and post-conditions are the same as described previously for executable methods.
- The body consists of a single return statement, which evaluates an expression and returns its value, necessarily of the same type as the function itself. The expression may recurse or invoke another function (or predicate, see next section)

The usefulness of functions arises from the fact that they can be invoked from specification statements. Their ability to recurse means makes them invaluable for specifying loop invariants, as each recursion of the function corresponds to an iteration of the loop.

Dafny is usually able to confirm termination of function recursion by itself. However, it may require help, in which case Veri-J will return a verification error stating that the function needs a variant. The variant must be placed after any pre- and post-conditions, as shown in the next example, which returns the product of all natural numbers in a range:

```
//@ FUNC
public static int product1(int j , int n)
//@ PRE: j >= 0 && n >= 0;
//@ VAR: n - j;
{
    return (j < n) ? j * product1(j + 1, n): j;
}
```

Veri-J allows functions to be called from compilable code as well as from specifications, so the function return statement must be written in valid Java code.

3.5.3 Predicates

Predicates are functions, albeit having a slightly different syntax, that return a Boolean. The predicate below checks that every element of an array is strictly positive:

```
//@ PRED              //line break required after PRED
public static void strictlyPos(int[] a)
//@ PRE: a != null;
{
    //@ forall k :: 0 <= k < a.length ==> 0 < a[k];
}
```

Predicate syntax has the following features:

- It begins with the Veri-J tag followed by the PRED keyword and a line break.
- The modifiers public and static are required
- The return type must be 'void' (Veri-J changes this to Boolean when translating to Dafny).
- The syntax and location of any pre- and post-conditions or variant are the same as described previously for executable methods.
- The body consists of a Boolean expression, which starts with the Veri-J tag and can contain any valid Veri-J or Java Boolean expressions. The expression may recurse or invoke another function or predicate.

Predicates can be invoked from specifications only, not from executable code. However, the same check can be implemented as a Boolean function if it is needed in executable code as well. For example, the following predicate and function both check whether an array is sorted:

```
//@ PRED
public static void sorted(int[] a)
//@ PRE: a != null;
{
    //@ forall m, n :: 0 <= m < n < a.length ==> a[m] < a[n];
}


//@ FUNC
public static boolean sorted(int[] a, int i)
//@ PRE: a != null && i == a.length - 1;
//@ VAR i + 1;
{
    return i <= 0 || a[i - 1] < a[i] && sorted(a, i - 1);
}
```

3.5.4 Lemmas

Lemmas are frequently used in the Reasoning sub-module. They are sub-proofs (theorems) that are used to break larger proofs up into smaller parts. The lemmas are proved independently and their results are used to prove the program as whole.

In Dafny (and Veri-J), lemmas are implemented as void methods. The method's pre-conditions and post-condition express the theorem. The theorem usually relates the method's in-parameters, if any, to each other. The code in the lemma's body proves the theorem. However, this proof is optional – Dafny simply assumes that the theorem is true if the code is omitted from the body.

To apply the lemma, a method-call statement is inserted in the code at a point where the pre-conditions hold and the post-condition needs to be established for the proof to continue.

The next example, showing the use of lemma *lemma1* in the proof of a program that calculates factorials, illustrates the required features of a lemma's syntax:

- It begins with the Veri-J tag followed by the LEMMA keyword and a line break.
- The modifiers public and static are required.
- The return type is void.
- The syntax and location of any pre- and post-conditions are the same as described previously for compilable methods.
- The lemma's body may be empty of executable code, in which case the theorem expressed by its pre- and post-conditions is assumed to be true.

A lemma's body can contain the same Java code constructs as a compilable method. However, if it is present then any such code must prove the lemma. Invariants, variants and mid-conditions can be used as required to do this.

Code in a lemma's body cannot declare reference-type variables or change the value of any reference-type variable that is passed to the lemma as a parameter. However, primitive-type variables can be declared and used as normal in the lemma's executable code.

Finally, regardless of their type, a lemma's in-parameters are immutable.

```
public class Factorial {

    //@ LEMMA
    public static void lemma1(int rez, int i)
    //@ PRE: i >= 0 && rez == product1(1, i);
    //@ POST: rez * (i + 1) == product1(1, i + 1);
    {
    }

    //@ FUNC
    public static int product1(int j , int n)
    //@ PRE: j >= 0 && n >= 0;
    //@ VAR: n - j;
    {
        return (j < n) ? j * product1(j + 1, n): j;
    }

    int factorial(int n)
    //@ PRE: n >= 0;
    //@ POST: r == product1(1, Pre(n)) && r >= 1;
    {
        int i = 0;
        int rez = 1;

        //@ INV: 0 <= i <= n && n == Pre(n)
            && rez == product1(1, i);
        //@ VAR: n - i;
        while(i < n)
        {
            lemma1(rez, i);
            i   = i + 1;
            rez = rez * i;
        }

        //@ MID: n == Pre(n) && rez == product1(1, n);
        return rez;
    }
}
```

3.5.5 Reads statement

Veri-J requires any non-array reference-type variables read by a function or predicate to be explicitly declared.[6] This is achieved by inserting a Reads statement after all other specification statements in the function declaration. The syntax of a Reads statement is :

```
//@ READS: identifier (, identifier)*;
```

For example, the Reads statement in Veri-J for a function that needed to read the field values of three instances a, b and c of some non-array object types would be:

```
//@ READS: a, b, c;
```

Use of the Reads statement for arrays or any primitive type variables causes an error.

3.5.6 Importing verification methods

The exercises set on the Reasoning sub-module frequently include functions, predicates and lemmas to be used in the solution. Many of these are used in more than one proof.

To facilitate this, Veri-J allows functions, predicates and lemmas to be imported from other Java files using Java's static import statement. The translator extracts the specified members and includes them in the translation to Dafny. Frequently used items can therefore be held in a common 'library' class from which they can be imported as required by each exercise.

The import statements are standard Java, so follow Java rules for their use. Items may be imported individually by name:

```
import static verij.testfiles.library.sorted;
```

or en masse

```
import static verij.testfiles.library.*;
```

The imported file must be in the same package (that is, folder) as the importing file. A number of the examples in the examples zip file use static imports.

3.6 Counter-examples

A counter-example is a set of values of parameters and variables which demonstrate that a proof is invalid. It is useful to have a counter-example when working out why a proof will not verify. The Z3 solver returns counter-examples to Dafny for unsuccessful proofs but unfortunately these are not passed on by the current version of the Dafny CLI. This feature is planned for a future release of Dafny and will be straightforward to implement in Veri-J when available. The final chapter of this report recommends this as an extension to the plug-in.

In the meantime, it is worth noting that the counter-example is available in the Dafny extension for Visual Studio Code (VSC). VSC can be accessed directly from Veri-J as described in section 3.1. The counter-example display in VSC is toggled with the F7 and F8 keys.

[6] As with the Modifies statement, the reasons for this relate to memory management by Dafny during the verification process and are outside the scope of this project.

Chapter 4

Implementation

This chapter describes the decisions I took to design and build an application that meets the objectives set out for it the introduction and the requirements described in the previous chapter. It is not a detailed guide to the code. Rather, it focuses on the key features of the application, the options available for implementing them and my reasons for choosing between these options.

4.1 Defining the task

The implementation can be broken down into a few key components:

- A formal grammar defining both the Java code and the specification annotations required for the Reasoning sub-module.
- A lexer and parser based on this grammar, including syntax error reporting
- A translator to translate the parsed code to Dafny.
- A mapping to associate translated Dafny code tokens to tokens in the input Java file, with a lookup facility to report verification errors.
- A user interface to run the translator and Dafny CLI and display the output of both of these to the user.

The narrative of this chapter follows the timeline of development of these tasks.

4.2 Program structure

An important consideration when designing the program was to separate its functionality into distinct, interdependent modules. This would make it easier to build, test and extend. The program's main requirements, in order, are to:

- read and parse the Java code, checking it for syntax errors
- translate the Java code to Dafny and build a Dafny-to-Java location map
- submit the translation to Dafny for verification

- retrieve the Java location of any verification errors using the location map
- display verification results in the UI

Figure 4.1 shows the structure of the program and the inputs and outputs to and from its major components.

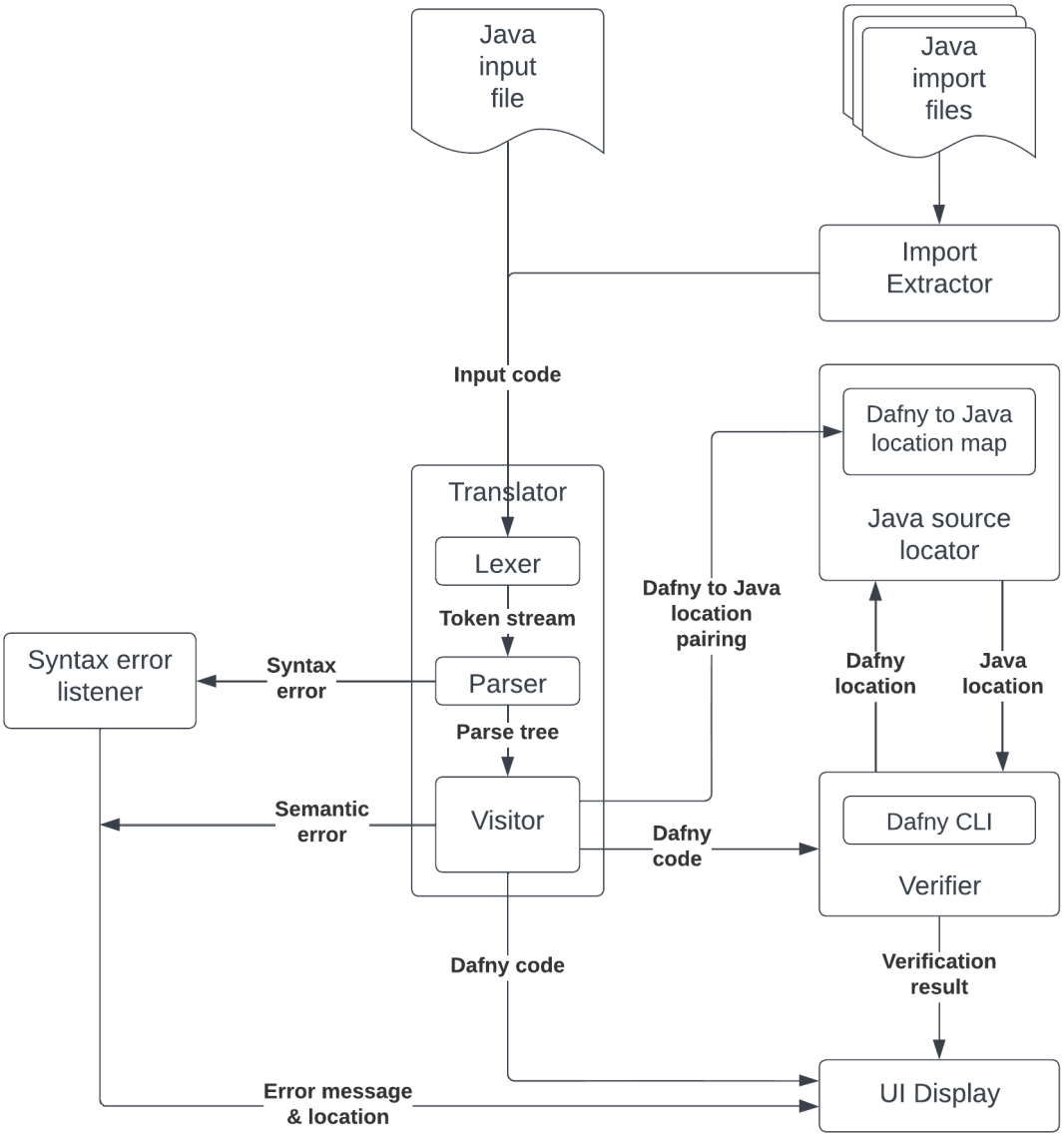


Figure 4.1: Program structure

4.3 Building a lexer and parser

4.3.1 Available options

Various well-established algorithms can be used for parsing code. However, programming a lexer and parser from scratch is a lengthy, complex undertaking that would neither have been practicable for this project nor added value to it. Therefore, I decided to use an existing parser or parser-generating tool for this purpose.

The most immediate option might have been to use IntelliJ's built-in parser, known as the Program Structure Interface or PSI. However, several factors militated against this choice. The PSI library is accessible only from within plugins running on IntelliJ, so any application relying on it would be inseparably coupled to this IDE. Furthermore, PSI's range of features and development environment are imperfectly documented and not extensive, making both initial development and future extension more challenging. Finally, the development community for PSI appears to be small, suggesting caution about its long-term viability as a development platform.

Numerous parser generators (known as metacompilers or compiler-compilers) are available and were explored for this project, such as YACC[32], JFlex[33] and CUP[34]. Some of these were separate tools for generating lexers and parsers only. Others offered limited functionality or were designed for specific domains. Many appeared to have fallen into disuse. However, one of them, ANTLR [1], stood out from the rest and was my choice for the implementation.

4.3.2 Choosing the ANTLR parser generator

ANTLR is a general purpose open-source metacompiler. Since its inception around twenty-five years ago, it has become perhaps the most widely-used solution for automatically generating language lexers and parsers. There were several compelling reasons for me to choose it:

- ANTLR generates the lexer and parser together as separate but well-integrated classes. During execution, both the character stream to the lexer and the token stream from it are fully accessible to the parser. This makes it possible for the parser to inspect a character or token and then use its literal value, rather than the token type, to determine which rule it should match.

- ANTLR can accommodate ambiguous and left-recursive grammars. It also has other useful features such as syntax error listeners, custom non-nodal fields[2], and the facility to embed executable code in the grammar to control parsing.

- ANTLR provides a well-thought-out GUI development environment as a plugin for IntelliJ. This has several features that make writing and debugging grammar rules easier and quicker. For instance, it displays an interactive parse tree that graphically depicts changes to the grammar as they are made. It also shows hints, warnings and error highlighting for the grammar as it is being coded in the editor pane.

- The ANTLR plugin is compatible with the Gradle build automation tool, which is recommended by IntelliJ for creating plugins for its IDEs.

[1] ANother Tool for Language Recognition.

[2] Custom fields of any primitive or reference type can be declared in the grammar for any rule. They are available in the corresponding node object in the parse tree.

4.4 Developing the grammar

4.4.1 Deciding where to start

Writing a new grammar from scratch application was unnecessary as a Java grammar is already available on ANTLR's GitHub repo[35]. However, the Reasoning sub-module course uses only a subset of Java's features. Furthermore, there are numerous features in Java that have no equivalent in Dafny.

This raised the question of whether it would be better to leave the existing grammar unchanged or to pare it down. The benefit of trimming the grammar is that any unsupported features in the input code would be detected and reported to the user as syntax errors rather than just being ignored. The downside, however, is that deleting any production rules could make the application less extensible in the future.

On balance, I concluded that the likelihood of unsupported features causing problems in use was small as the range of Java features used on the Reasoning sub-module is well-defined. Maintaining extensibility was more important as both the module requirements and Dafny's capabilities could evolve in the future. Using the available grammar without editing it would mean that further Java constructs could be included in Veri J simply by overriding existing methods from the auto-generated classes, rather than having to extend the grammar and regenerate the translator each time.

4.4.2 Writing new rules

The grammar needed new rules to derive the verification statements described in Chapter 3. These rules in turn derive new expressions of three types: reference expressions such as array slices; integer expressions such as array aggregates; and Boolean expressions such as comparison chains. For example, the new rule for preconditions, requiring a Boolean expression is:[3]

```
rapjPrecondition
    : RAPJ_TAG PRE COLON rapjBooleanExpression SEMI
```

All the new rules needed either to begin with a '//@' tag, or to be derived only from one which did. This ensures that none can be derived from any Java rule in the grammar.

The new expressions needed to be able to derive any of several Java expressions. For example, the bounds of an array slice could be any Java expression that evaluates to an integer. Similarly, a quantifier statement might need to include a variety of Boolean and integer expressions. However, there are also many Java expressions such as bit-shift and lambda expressions, that are not supported in Dafny. Allowing them to be translated would cause verification errors. There were three ways I could address these conflicting requirements:

1. The first would be to replicate the required Java expressions and derive the copies from the verification statements. The benefit of this is that unwanted expressions would be excluded from the new statements. However, the duplication required would make the code less understandable and more complex to extend in future.
2. The second option was to delete illegal expressions from the existing Java grammar. However, this was undesirable for reasons already discussed in section 4.1.
3. The third option was simply to allow verification statements to include all existing Java expressions as well as the new ones. I chose this option because it avoids duplication and makes the new rules simpler. For example, the rule for an array slice is:

[3] I gave all new rule variables a prefix to distinguish them from existing (Java) rules. The prefix used here is 'rapJ'. This was the initial working name for the plugin. It was an abbreviation of 'Reasoning about Programs in Java'.

```
rapjArraySlice
    : identifier lEncl=(LBRACK | LPAREN)
              lBound=expression? RANGE uBound=expression? rEncl=(
                  RBRACK | RPAREN)
    ;
```

The bounds of the slice can be any Java expression. Although this does not detect Java expressions that are not supported by Dafny, any such expressions are identified and reported by Dafny, which then ends without invoking Z3. As the chapter on evaluation shows, the performance hit of this is negligible.

A further consideration when writing rules for the new expressions was that a translator's output is not affected by operator precedence. It does not matter in what order the sub-expressions embedded in a composite expression are translated – the translation will always be the same.

This matters because maintaining operator precedence in ANTLR grammars requires each precedence level to have its own alternative body within the rule. The alternatives are placed in order of operator precedence. The parser matches the first rule it encounters. For instance, the alternative for multiplying or dividing two expressions would be listed above the alternative for adding or subtracting them. This is necessary if the expressions are being parsed for execution or evaluation. However, for a translation, the implementation can be simplified by including all binary operators as alternatives between the two expressions within a single rule:

```
rapjBooleanExpression
    : ...
    | rapjBooleanExpression bop = (AND | OR | EQUIV | IMPL |
       RIMPL) rapjBooleanExpression
    | ...
    ;
```

In the rule above, operators in the `bop` subrule have different precedence but for a translator can be placed together rather than having to be being split across several alternative rule bodies.

4.4.3 Ambiguous tokens

When writing the new rule for array slices, I needed to introduce a new double dot 'range' operator. However, if the lower bound of the slice was an integer literal (with no trailing space), then the lexer would incorrectly match the integer and the dot that followed it to a float literal. Also, three consecutive dots needed to be matched to an ellipsis rather than to a range operator and a dot. To achieve this, the parser needed to be told to look ahead and match float literal and range tokens only if they were not followed by a dot. I used semantic predicates in the lexer grammar to do this.

Semantic predicates in ANTLR grammars are expressions written in the language being parsed that evaluate to true or false. They can be inserted anywhere in the body of a parser or lexer rule. The rule will be matched only when the predicate is true.

Predicates are able to read the input stream via the `_input` object. The methods `_input.LA(n)` and `_input.LT(n)` can be used to read characters or tokens respectively, either ahead of `(n > 0)` or before `(n < 0)` the current read point.

I used the predicate `{_input.LA(1)!= '.'}?` to prevent a dot being matched as a DOT token if it is followed by another dot:

```
DOT:  '.' {_input.LA(1) != '.'}?;
```

A similar predicate in the RANGE lexer rule prevents it from matching the first two dots of

an ellipsis:

```
RANGE:  '..'{_input.LA(1) != '.'}?;
```

4.4.4 Splitting statements

Some statements in Java need to be translated into multiple statements in Dafny. For example, as Dafny does not support unary increment/decrement operators, any embedded inc/dec expression in Java results in an extra statement above (prefix) or below (postfix) the translation of the containing statement. So a statement such as, say, `a = ++i;` in Java would become:

```
i := i + 1;
a := i;
```

To achieve this split, the visitor to the sub-node (here, the inc/dec expression) needs to reach back up the parse tree and insert the new statement before (prefix) or after (postfix) the containing statement. This requires that there be somewhere for the extra statement to be placed and held until it is added to the output. I used ANTLR 'local variables' to provide such a place.

ANTLR local variables can be declared for any rule in the grammar and can be of any type. ANTLR adds them as non-nodal fields to the node class it generates for the rule in the parser. I declared a custom `PlaceHolder` type for use as a placeholder before or after the return value of statement `visitors`. I then annotated the statement rule in the grammar with two local variables, `preText` and `postText`, of `PlaceHolder` type:

```
statement
locals [PlaceHolder preText; PlaceHolder postText;]
: ...
| IF parExpression statement (ELSE statement)?  #Statement03_if
| ...
| RETURN expression? ';'                #Statement11_return
;
```

`preText` and `postText` appear as fields of the node object of all `statement` nodes in the parse tree. The `statement` visitor adds any text placed in these fields to the start or end its return value.

4.4.5 Trapping keywords

The use of Dafny, Java or new Veri-J keywords as identifiers anywhere in the code submitted to Dafny would return a verification error. To address this, I replaced all occurrences of the `IDENTIFIER` terminal node in the parser grammar with a new `identifier` variable and added a new parser rule for it:

```
identifier
: IDENTIFIER
;
```

This change introduced a visitor method for the identifier, which is used to prevent verification from being attempted if any identifier in the input code matches any item from a list of reserved words saved in the plugin.

4.5 Implementing the translator

4.5.1 Generating base classes

As well as a lexer and parser, the ANTLR tool[4] generates a default listener class and a default visitor class for use as base classes by the application. The default listener contains empty enter and exit methods, and the default visitor an empty visit method, for each parser rule in the grammar. Application classes can extend either or both of these default classes.

4.5.2 Choice of listener vs visitor

Listener events are void methods. This means that additional data structures are needed in the application to propagate output between nodes. One approach to this is to use a field or global variable. Another is to add local variables to the grammar, as described in section 4.3.4, to store output values in nodes. In both cases, the task of producing the output is shared between enter and exit events.

By contrast, visit methods do return values, which travel up the tree from the lower nodes to the root. Usefully, in a visitor pattern, the order in which overridden child nodes are visited is defined when writing the code rather than being fixed as in the listener. Furthermore, the result for each node is produced entirely within the visitor method rather than being split across two events.

Although a listener-based implementation would have worked for this application, its needs for custom data propagation and the split across two events for each node are awkward and would have added unnecessary complexity. Therefore, I chose to base Veri-J on a visitor pattern.

4.5.3 Overall approach

I implemented the translator by extending the base visitor class (with derived class `Visitor`) and overriding those of its methods that were needed in the translation. The translation is assembled in string form in each visitor, using the values of its terminal nodes and the return values from the visitors to its sub-nodes. The visitor to the root node, `CompilationUnit`, returns the complete translation.

According to Terence Parr, the lead architect of ANTLR since its inception [36]:

> Translation involves fully understanding each input phrase, picking an appropriate output construct, and filling it with elements from the input model. Trying to cleverly replace input symbols with output symbols rarely works well. You end up with what we call a literal translation in a natural language. For example, faire un canard in French means literally "to make a duck." The real translation, though, is to "hit a wrong note."

Despite this, the translation was mostly straightforward as executable code in Java is reasonably similar to that in Dafny. Furthermore, the syntax used for proof statements on the Reasoning sub-module is similar to the syntax used in Dafny. Nevertheless, some parts of the translation did require particular attention and these are described next.

[4] org.antlr.v4.Tool

4.5.4 Array slicing and concatenation

Array slicing and concatenation required additional rules in the grammar:

```
rapjArraySlice
: identifier lEncl=(LBRACK | LPAREN)
lBound=expression? RANGE uBound=expression?
rEncl=(RBRACK | RPAREN)
| PREVALUE LPAREN rapjArraySlice RPAREN
;

rapjArrayConcatn
: rapjArraySlice (COLON rapjArraySlice)*
;
```

Array slices are not implemented as arrays in Dafny. Rather, it converts them to sequences. A Dafny sequence can be 'converted' to an array only by replacing it with a new array, as shown in the Dafny code below:

```
method SeqToArray <T> (s: seq<T>) returns (a: array<T>)
   ensures fresh(a) && a[..] == s
   {
      a := new T[|s|](i requires 0 <= i < |s| => s[i]);
   }
```

However, such conversion is not available in the pre-compile 'ghost space' occupied by specification statements and ghost items (functions, predicates and lemmas). This is because Dafny does not allow anything in the ghost space to write to heap memory. So if an array slice (or concatenation of slices) needs to be passed to a lemma, function or predicate as a parameter, the parameter must be of sequence type in the callee's signature. Any attempt to pass a sequence to an array-type parameter (or vice versa) results in a Dafny error.

However, unlike elements in Dafny arrays, elements in Dafny sequences are immutable. This does not matter in ghost items as they cannot write to memory, so anyway cannot update values in any arrays that are passed to them. However, the immutability of sequences does mean that sequences cannot be used instead of arrays in executable code, where array elements frequently do need to be updated.

Veri-J therefore needs to treat executable methods differently from ghost items in relation to arrays. Arrays in executable methods are retained as arrays in the Dafny translation but arrays in ghost items are translated to sequences.

Another distinction is that arrays, being a reference type, can be null whereas sequences, being a value type, cannot. In Dafny, any expression that asserts that a sequence is not null results in an error. However, expressions asserting that arrays are not null are frequently used in the Reasoning sub-module. To accommodate this, Veri-J translates such expressions in Java to `true` if the array is being translated to a sequence.

To summarise, an array in Java needs to be treated as an array in Dafny executable code (in case its element values need to be changed) but as a sequence within Dafny ghost items (in case they are passed an 'array' slice as a parameter).

This means that when translating method calls from Java, Veri-J needs to distinguish between calls to ghost items and calls to executable methods. It passes array-type parameters as sequences to the former and as arrays to the latter.

Veri-J implements these requirements by visiting the parse tree twice. The first pass does not return a value. Rather, it populates a list with the names and input parameter details for all ghost items in the target Java code, including imported ones. The second pass produces the Dafny translation. It uses the list to check whether any of the methods called are ghost items and to

identify their array-type in-parameters (if any), which it changes to sequences.

For example, if the types of in-parameters a, b and c of `myMethod`, are of primitive types, one-rank array and multi-rank array respectively, then the translation of a call to `myMethod` would be `myMethod(a, b[..], c)` if `myMethod` was a ghost item but `myMethod(a, b, c)` if it was an executable method. Appending slice notation `[..]` to the identifier `b`, converts it to a sequence in Dafny.

Also during the second pass, expressions of the form `id != null` are translated to true and of the form `id.length` are translated to `|id|`[5] if `id` is the name of a sequence contained within a ghost item.

4.5.5 Array aggregate functions

Specification statements used on the Reasoning sub-module may refer to the aggregate product or aggregate sum of the elements of an array[6] or array slice. As described in section 3.4.10, Veri-J uses the key words `Pi` and `Sigma` for these. To accommodate these, the grammar requires an extra alternative in the additional expressions rule:

```
rapjIntExpression
   : ...
   | fn=(PI | SIGMA) rapjArraySlice
   ;
```

For example, the Veri-J invariant:

```
//@ INV: res == Sigma a[..i);
```

asserts that the variable res is equal to the sum of all elements of array `a` from `a[0]` to `a[i-1]`.

Two functions coded in Dafny, `arraySum` and `arrayProd`, which calculate the aggregate sum and product respectively, have been hard-coded into Veri-J. The visitor for the rule above returns a method call in Dafny to whichever of these functions is needed. and the function's code is appended to the translation. One of these Dafny functions, `arrayProd`, is shown below:

```
function arrayProd(a: array<int>, m: int, n: int) : int
requires 0 <= m <= n;
requires n < a.Length;
reads a;
{
    if m == n then a[n] else a[n] * arrayProd(a, m, n - 1)
}
```

4.5.6 Entry-state or 'pre' values

The pre-value of a method parameter is its value upon entry to the method. As described in section 3.4.11, this is denoted using a 'pre' subscript on the Reasoning sub-module and with a built-in `Pre()` function in Veri-J. The `Pre()` function in Veri-J can be applied to primitives, arrays, array slices or array elements. For example, the expressions x_{pre}, $(a[3])_{pre}$ or $(b[i..j))_{pre}$ in module notation become `Pre(x)`, `Pre(a[3])`, and `Pre(b[i..j))` respectively in Veri-J notation.

[5] The size of Dafny sequences is denoted by cardinality bars.

[6] In this case, module notation always denotes whole arrays as slices, e.g., array a would be `a[..)`.

The grammar rule needed for the Pre() function is:

```
rapjReferenceExpression
    : PREVALUE LPAREN (rapjArraySlice | expression) RPAREN
    ;
```

In Dafny, the entry-state value is obtained using the built-in `old()`[7] function. However, the implementation of this function can access pre-values for memory accessed by reference (heap memory) only. It does not access stack memory and so cannot provide the pre-values of primitive-type in-parameters[8]. Application of the `old()` function to a primitive-type parameter is redundant and ignored by Dafny. Use of primitive-type in-parameters in postconditions always refers to their entry-state value[9].

As shown in the Java and Dafny code for the `dodgeyDouble` method below, I have implemented pre-values for primitive-type in-parameters in Veri-J by using 'shadow' variables in the Dafny translation. The method declaration visitor inserts a declaration of a shadow variable for each primitive-type in-parameter in each executable method. Also, the name of these in-parameters in saved in list field `primitiveMethodParameters` of the `Visitor` class. The values are maintained in the list only for the duration of the method-declaration visit.

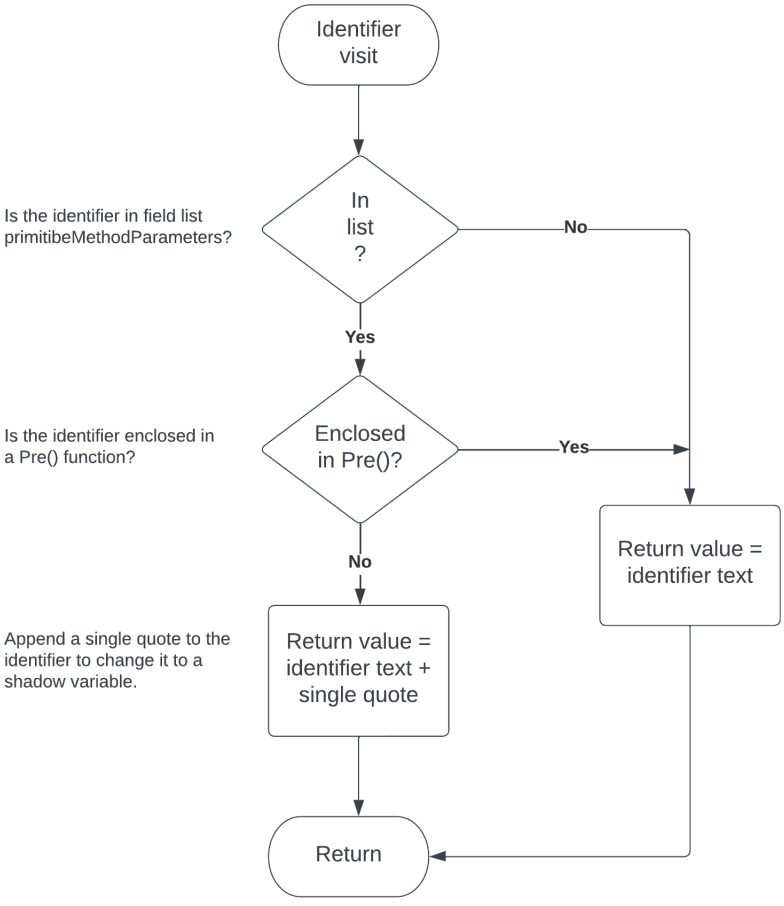


Figure 4.2: Implementation of the `Pre()` function in the identifier node visitor

[7]The semantics of old() in Dafny conflict with the 'old' value used on the Reasoning sub-module, which refers to the value of an in-parameter in the current-but-one state. The module 'old' value is needed only in proofs and has no equivalent in Veri-J.

[8]The reason for this is undocumented.

[9]To avoid confusion on this point, Dafny makes the values of primitive-type input parameters immutable by default. It allows this restriction to be partially removed by allowing the declaration of local variables with the same names as in-parameters. However, such synonyms remain separate in memory from the in-parameters that share their names. Postconditions can still only access the (immutable) value of the in-parameter.

Shadow variable identifiers are comprised of the in-parameter name followed by a single quote. The single-quote character is allowed in identifiers in Dafny but not in Java. Its use here avoids any possibility of duplication with a user-defined name.

The rest of the implementation is contained within the identifier node visitor, as shown in Figure 4.2.

The features of the implementation described above can be seen in Veri-J's translation of the `dodgeyDouble()` method (module Week 7 tutorial, Question 2).

The Java code for `dodgeyDouble()` is:

```
int dodgeyDouble (int x)
//@ PRE: true;
//@ POST: r == 2 * Pre(x);
{
    //@ MID: x == Pre(x);
    int y = x;
    //@ MID: y == x && x == Pre(x);
    x = x + 1;
    //@ MID: y == Pre(x) && x == Pre(x) + 1;
    y = y + y;
    //@ MID: y == 2 * Pre(x);
    return y;
}
```

Its translation to Dafny by Veri-J is:

```
method dodgeyDouble(x: int) returns (r: int)
requires true;
ensures r == 2 * x;
{
     var x' : int := x;  //declaration of shadow variable
   assert x' == old(x);
   var y : int := x';
   assert y == x' && x' == old(x);
   x' := x' + 1;
   assert y == x && x' == old(x) + 1;
   y := y + y;
   assert y == 2 * old(x);
   r := y;
   return r;
}
```

4.5.7 Quantifier expressions

A quantifier expression states that a given Boolean expression is true either universally for all, or existentially for at least one, combination of some variables, known as 'bound' variables. They are widely used in specification statements on the Reasoning sub-module using the universal ($\forall$) and existential ($\exists$) symbols. Dafny and Veri-J support quantifiers with the use of the `forall` and `exists` keywords. The structure of the Veri-J grammar rules for quantifiers are:

```
booleanExpression
:  QUANTIFIER boundVariable (',' boundVariable)* '|'
    booleanExpression
'::'booleanExpression;

boundVariable:
: (primitiveType ('[' ']')?)? identifier;
```

For example:

```
forall int x, long y | 0 <= x <= 5 && y <= 3 :: x * x * y <= 75;

forall a :: 2 <= a ==> (exists b :: a < b < 2 * a);
```

The expression after the double colon (::) must be true for the entire expression to be true. It can be any Boolean expression recognised by Veri-J. The first example illustrates use of the optional type (`int` and `long`) and optional domain `(0 <= x <= 5 && y <= 3)` for bound variables x and y. The second example illustrates nesting of quantifier expressions.

4.5.8 Imports

As described in section 3.5.6, commonly used specification methods (i.e., lemmas, functions and predicates) can be saved together in a single Java class and then imported using Java import declarations in the files that need them. The code of the imported methods is extracted, translated to Dafny and added to the rest of the translation before verification.

This functionality is implemented in the import declaration visitor method in the `Visitor` class. The method parses and visits the imported file with a new instance of the `Visitor` class, which returns the Dafny translation of the items to be imported. This translation is then added to the main translation before it is submitted for verification. This approach could also process any import declarations it finds in the imported files themselves, but such cascading has been blocked to avoid confusing the user.

The visitor to an imported file will collect the translation only for the method named in the import declaration (or for all methods if the wildcard '*' is used).

4.5.9 Edge cases

Increment and decrement expressions

In Java, the increment ('++') and decrement ('--') unary operators are used to increase or decrease the value of a variable by one. Both have prefix and postfix forms. When such an expression is used by itself as a standalone statement, the prefix and postfix forms have the same effect.

However, when the expression is embedded in an assignment expression, operator precedence comes into play. With the prefix form, the variable value is changed first and the changed value is used in the assignment. By contrast, with the postfix form the variable's existing value is used in the assignment and the variable's value is changed after the assignment has been made. For example:

```
i = 3;
a = i++; // after postfix: a = 3, i = 4
b = ++a * 2; // after prefix: b = 8, a = 4
```

Dafny does not support unary increment or decrement operators. However, as they are a convenient feature of Java, I explored how they could be translated to Dafny. The required translation of the typical usages described above is shown in Table 1:

Java code example	Dafny translation
`i++;` `++i;`	`i := i + 1;`
`a = i++;`	`a := i;` `i := i + 1`
`a = ++i;`	`i := i + 1;` `a := i`

In these examples, the embedded inc/dec expressions simply require an extra statement before (for prefix) or after (for postfix) the original statement. In principle, the additional statement could be inserted by 'back-placing' it from the inc/dec expression visitor method into the custom `pretext` or `postText` statement node fields described in section 4.3.3. The statement visitor would then add the values of the `preText` and `postText` fields to the start and end of its return value.

However, although such an implementation would work correctly most of the time, it would not correctly translate many cases where the incremented or decremented variable is repeated in the same statement. For example:

Java code example	Incorrect translation	Correct Translation

`a = i + (++i);`	`i := i + 1;`
	`a := i + i;`
		`a := i;`
		`i := i + 1;`
		`a := a + i;`

`a = (i++)+ i;`	`a := i + i;`
	`i := i + 1;`
		`a := i;`
		`i := i + 1;`
		`a := a + i;`

The increment or decrement expression affects nothing to the left of its operator but applies to all instances of the variable to the right.

For a correct translation, the Java input statement needs to be deconstructed and rebuilt in Dafny as a sequence of assignments of its constituent expressions. The translated increment expression needs to be inserted at the point in the sequence where it takes effect. The table illustrates the simplest possible examples but the input statement could have subexpressions nested to an arbitrary number of levels.

Although such statements would be unusual in practice, they are allowed by Java and so could be encountered by the plugin. The implementation must either correctly translate them in all cases or disallow their embedding. I decided that within the time available, it would be impracticable to design, build and test a solution for this feature. Veri-J therefore disallows inc/dec expressions unless they are standalone statements.

However, a solution if implemented in future would most likely require back-placing of the output statement as described in section 4.3.3 I have therefore included the `preText` and `postText` sequence node class fields and a shell definition of the `PlaceHolder` class in the implementation to facilitate future extension.

Method calls

The Dafny specification stipulates that an executable method call must be either a standalone statement or comprise the entire right-hand side of an assignment. In other words, an executable

method call in Dafny cannot be contained in any expression other than itself or an assignment.[10]

At first glance, the translations of typical usage of method calls could be as follows:

Java code example	Dafny translation
`calc(2, 5);`	`calc(2, 5);`
`int i = calc(2, 5);`	`var i: int := calc(2, 5);`
`int i = x + calc(2, 5);`	`var varID := calc(2, 5);` `var i: int := x + varID;`

However, although the first two translations are correct, the third is unsafe. This is because `calc()` may have side-effects that change the value of x, so the value of x must be assigned to i before `calc()` is invoked.

We can see that the correct implementation of method-call translation would be similar to that of the increment/decrement expression described in the preceding section. I decided to restrict it for the same reasons as before. Therefore, Veri-J disallows use of executable method calls that are not standalone or that do not comprise the entire right-hand side of an assignment.

4.6 Verification and integration with Dafny

Verification requires Veri-J to run the Dafny CLI, with the location and name of the translated file supplied as parameters. Veri-J implements this by using a Java `ProcessBuilder` class, which is available to run operating system commands. Feedback from Dafny is collected from the `ProcessBuilder` into a Java `BufferedReader`, the content of which is then read line-by-line. Regular expressions are used to extract feedback constructs from each line, such as error coordinates and messages, which are then processed as appropriate.

Implementing the call to the Dafny CLI raised the question of how it should be integrated with the plugin. One approach was simply to include the Dafny executable and all its dependencies within the plugin, to be downloaded and installed together. The second option was to require the user to install Dafny separately and provide the facility for it to be linked to Veri-J. I chose the second approach because:

- Dafny is an ongoing project. New patches, updates and versions are released regularly. Packaging Dafny in Veri-J would make it harder for the user to access the latest features.
- Separate installation of the plugin and Dafny makes the cause of any problems during installation easier to identify.
- It makes clear to users from the start that Veri-J and Dafny are distinct applications. Having to visit Dafny's GitHub site will introduce users to Dafny and make them aware that they can use it independently of Veri-J.

Linking Dafny to Veri-J requires users to select and save the location of the Dafny executable in the plugin. This is a one-off requirement that is performed from the UI during installation of the plugin.

[10] This restriction does not apply to calls to specification functions and predicates. These are non-executable and can be called freely from within specification statements, functions, predicates and lemmas.

4.7 Error handling

Veri-J has a three-tiered approach to processing errors. Syntax errors are reported first, followed by semantic errors and then verification errors. Errors from one tier must be cleared before any from the next are reported.

This progression arises naturally from the sequence of execution of the different parts of the application. Syntax errors are detected while the parse tree is being built, using an error listener provided by ANTLR. Semantic errors are trapped by checks in the visitor methods during the parse tree visit. Verification errors are raised by Z3 after when it attempts verification of the code submitted to it by Dafny.

In all cases, Veri-J reports the Java coordinates (line number and position) and description of the error to the user in its feedback pane. The location of the error is highlighted in the Java code editor.

4.7.1 Syntax errors

Syntax errors are generated by the parser when it is unable to match the token stream to any of the rules in the grammar. Veri-J redirects information about such errors from the parser to the UI with a custom error listener, `SyntaxErrorListener`, that extends the ANTLR API's `BaseErrorListener` class. The base class has a `syntaxError()` method that receives the error location and message from the parser. Veri-J's implementation of this method displays these details in the UI and highlights the error location in the input file.

To provide live syntax error checking I declared two custom keystroke listeners – `TypingHandler` and `BackspaceHandler` - that extend the base classes `TypedHandlerDelgate` and `BackSpaceHandlerDelegate` in the IntelliJ plugin SDK. These base classes have methods `charTyped` and `charDeleted` that are invoked whenever a character is typed or deleted.

At each keystroke, the custom implementation of these methods instantiates the ANTLR parser, assigns a custom `SyntaxErrorListener` to it, parses the code being typed and displays error details, if any, in the UI.

4.7.2 Semantic errors

Comprehensive semantic checking by Veri-J would be redundant because it is already implemented in Dafny. Therefore, Veri-J's translator reports semantic errors in two types of situation only. The first is when the error is specific to Veri-J, so would not be detected by Dafny. The second is when the check needed to detect the error already occurs as part of the implementation or is close to it, in which case the error can be reported opportunistically without increasing the complexity of the code. Veri-J relies on Dafny for all other semantic checks.

The checks included in Veri-J are:

- Disallow use of Java, Dafny or additional Veri-J reserved words.
- Disallow use of 'r' as an identifier outside method postconditions.
- Disallow use of increment/decrement expressions unless they are standalone statements.
- Disallow use of executable method calls that are not standalone or that do not comprise the entire right-hand side of an assignment
- Disallow use of Java for loops and switch statements.

- Disallow identifiers that start with an underscore, end with an apostrophe or contain a '$' sign.

- Various other checks to disallow use of unsupported constructs and operators.

These checks are implemented in the visitor methods of the relevant nodes. When an error is detected, its location and description are saved in a `JavaTokenLocation` object, which is then added to a `semanticErrors` list field.

Detection of a semantic error does not halt the translation. Instead, after the entire parse tree has been visited, the `semanticErrors` list is checked. If it contains errors, their details are displayed in the UI. Execution then terminates without the translation being displayed or submitted to Dafny.

4.7.3 Verification errors

Verification errors are raised by Z3 if it is unable to prove that the code submitted to it by Dafny is correct. The term 'verification error' is also used here loosely, to include semantic errors detected by Dafny before verification is attempted. In both cases, Dafny passes the coordinates and a description of the error back to the calling code.

The calling code matches the Dafny error coordinates to the coordinates of the corresponding token in the Java input code, highlights the Java token in the code editor and displays its coordinates and the description of the error in the feedback pane.

The current version of Dafny raises verification errors only for expressions in verification statements (pre-post- and mid-conditions, variants and invariants) and also the return statement. This raised the question of whether Veri-J's verification error reporting should cover the entire program being verified or only these constructs.

However, development of Dafny is ongoing, so its error reporting may become more extensive in future. Furthermore, non-verification errors, which are not generated by Z3, could arise unexpectedly from any location in the Dafny code. For these reasons, I decided that the entire Dafny translation should be mapped to the Java code.

Veri-J does this by mapping the (Dafny) location of every character in the translation of a Java keyword, operator or identifier token back to the (Java) location of that token. The translations of minor tokens such as brackets are not explicitly mapped. However, the map is sorted, so if Dafny returns an unmapped error location, the Java token nearest to it can still be retrieved.

Error location objects

The location of each Java token included in the map is stored in a `JavaTokenLocation` object. The coordinates of the most-recently-added character in the Dafny translation are maintained by fields `currentLineNo` and `currentCharPosn` of the `JavaSourceLocator` class. The position of each Dafny character being mapped is stored in a `DafnyCharacterLocation` object.

Error location mapping

The `DafnyCharacterLocation` and `JavaTokenLocation` objects are saved as key-value pairs in tree map `errorlocations`, which is a field of `JavaSourceLocator`. Because each Java token usually gives rise to several characters in the Dafny output, each `JavaTokenLocation` value is typically mapped from multiple `DafnyCharacterLocation` keys.

The mapping is implemented as the translation takes place by methods `alm()` and `icp()`[11] of `JavaSourceLocator`. The `alm()` method maps Dafny character locations to their corresponding Java token locations whereas the `icp()` method merely increments the character position.

For example, the return statement for the array slice `Pre()` visitor method is:

```
return sl.alm(ctx.PREVALUE(),"old(") + visit(ctx.rapjArraySlice
    ()) + sl.icp(")");
```

This example would return translate an input, say, Pre(a[..)) to old(a[..]). Method alm() maps the location of each character in the Dafny snippet old(to the location of the Java token for Pre. Method icp() does not map the closing bracket of old() but only increments the character position. Mapping of the a[..) argument would be implemented separately, within the array slice visitor.

Error retrieval and lookup

The calling code uses regular expressions to retrieve error coordinates and messages in the feedback received from the Dafny CLI. Error coordinates are passed to method `getJavaLoc()` of `JavaSourceLocator`, which retrieves the location of the corresponding Java token from the `errorLocations` list and returns it to the UI.

If the error coordinates returned by Dafny are not found in the list, `getJavaLoc()` uses the mapped Dafny coordinates immediately before the ones returned. Alternatively, if these are not on the same line of the translation, it uses the mapped Dafny coordinates immediately after the ones returned.

For this approach to work, it is necessary that the map should maintain the Dafny location keys in ascending order of line number and character position. This is achieved using a Java TreeMap. TreeMap requires that the key type used should implement Java's `Comparable` interface and its `compareTo()` method. The implementation of `compareTo()` in the `DafnyCharacterLocation` class specifies that key ordering should be first by line number and then by character position.

Verification error mapping for imported files

As the import declaration rule appears before the class declaration rule in the Java grammar, Veri-J visits any imported members (functions, predicates, lemmas and executable methods) before the main body of code. Dafny does not require methods to be placed within classes, so Veri-J places the imported translations at the start of the output, before the class declaration.

Imported members use the same fields `currentLineNo` and `currentCharPosn` to map locations in the Dafny translation. The `currentLineNo` field maintains an unbroken line count through all imported files and the main input file, for the entire Dafny translation.

However, continuity of line number is not available for Java locations because they are obtained from the coordinates of the Java tokens, which are specific to each file. Therefore, to avoid confusing the user, verification errors that relate to imported files need to include the file name. To achieve this, the `JavaTokenLocation` class has a `filename` field that holds the name of the file in which the error was found. The file name is displayed in the UI only for verification errors located in imported files, not for those in the main input file.

[11] The method names 'alm' and 'icp' stand for 'add location mapping' and 'increment character position' respectively.

4.8 User interface

4.8.1 Design principles

The decisions I took when designing Veri-J's user interface were as much about users' needs and how the plugin should meet them as they were about technical issues.

The primary purpose of the plugin is to be a study aid on the Reasoning sub-module. However, I felt that with an appropriately designed UI, the plugin could also be an introduction to Dafny and wider application of formal methods.

My goal was that the plugin should deliver an experience that facilitated these outcomes. In particular, I aimed for a design that allowed users to see a clear distinction between Veri-J and Dafny.

For example, the Dafny CLI is installed separately from Veri-J and then linked to it rather than being contained within Veri-J's installation zip file. This makes the separation between the two applications clear to the user from the start.

Also, Veri J's tool window not only displays the Dafny code generated, it also allows the code to be opened directly from the UI in Visual Studio Code. This will allow users to learn Dafny by comparison with the Java they have just written and perhaps encourage them to explore it directly for themselves.

Differing source labels for error messages also make this distinction clear. Those from the translator are labelled as syntax or semantic errors, whereas those from Dafny are labelled as verification errors.

4.8.2 Minimalist vs modular design

A minimalist look would add as few new components to IntelliJ as possible by using IntelliJ's existing UI elements instead. The only new component would be an extra item in the main menu. This would open a submenu to run verification and other actions needed. Feedback and error messages from the plugin would appear in IntelliJ's built-in 'Run' tool window. The Dafny code generated would appear in a tabbed editor pane, alongside the editor pane for the Java input code being verified.

The alternative to the minimalist look would be to design the UI as a self-contained module that contained all the components needed, for use exclusively by the plugin. This would entail adding a custom tool window (i.e., a tabbed edge panel) to IntelliJ. The tool window would be split into two panels, one to display feedback and the other for the Dafny translation. A side toolbar would provide access to the actions needed. The tool window could be minimised when not in use or closed altogether by deselecting it in IntelliJ's 'View' menu.

I chose the second of these options. Although the first option avoids adding to IntelliJ's already-packed visual environment, it makes the plugin's functionality less obvious to the user. As there are no dedicated components, users would need to know where to look for verification results and how to tell them apart from routine messages from the Java compiler. By contrast, the modular design presents all actions and result displays in one place. Users can be confident that everything relevant to Veri-J is in the tool window and everything outside it is extraneous.

4.8.3 IntelliJ plugin development SDK

IntelliJ's plugin development API contains base classes for tool windows, menu actions and keystroke listeners. Methods in these classes provide access to the current project and components

such as code editor panes, menus and the internal code parser.

The base classes are extended by the plugin's custom classes to provide the functionality needed by the custom UI. These derived classes are registered in the plugin.xml start-up file. Registered classes are instantiated at start-up by IntelliJ runtime API.

4.8.4 Running Veri-J

The plugin is run by clicking the Verify button in its toolbar. The implementation code reads the Java code from the current editor and uses the custom lexer, parser and visitor classes to translate it to Dafny. If there are syntax or semantic errors, the location and description of these is displayed and the program terminates. Otherwise, the implementation passes the translation to the Dafny CLI and parses the feedback. If there are verification errors, it invokes `getJavaLoc()` as described in section 4.7.3 to obtain the locations of the errors in the Java input file and reports them to the user. If there are no verification errors, it reports success.

4.9 Test-driven development

Throughout the testing process, I used Mockito to mock the interactions between modules to allow me to test them in isolation from each other. This allowed my test to be very specific to the tested module, and for me to be sure that any errors found were caused by the code I was currently writing.

4.9.1 Testing the translator

I used the JUnit unit testing framework to build and test the code for Veri-J's translator step-by-step. At each step of the build, I wrote and tested the code needed to translate an input code construct, such as a precondition or class declaration, to Dafny. I wrote two text files for the test at each step – a Java input file and a file containing its expected Dafny translation. The tests compared the expected and actual outputs.

I planned the build process progressively, using the Reasoning sub-module tutorial exercises in chronological order. This ensured that the translator code for all the exercises was tested.

Chapter 5

Evaluation

This chapter describes how Veri-J was assessed for usability and performance (speed) after the initial build and testing had been completed. The purpose of the assessment was to establish the extent to which the plugin met its objectives, identify and correct any bugs or omissions and help determine how it could be extended in the future. The chapter ends with a reflection on the evaluation findings.

5.1 Usability assessment

5.1.1 Methodology

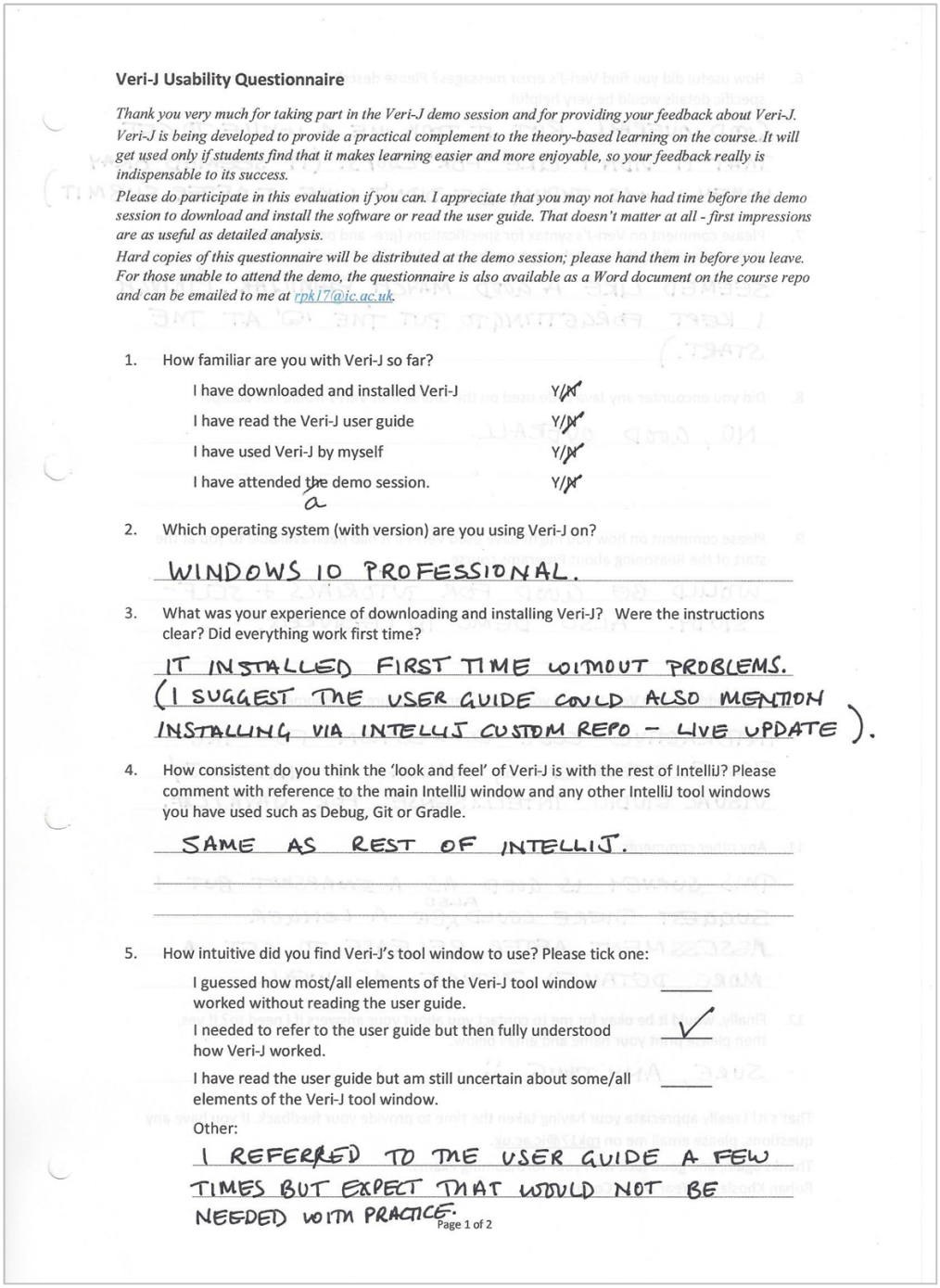
Veri-J will succeed only if students find that it makes learning easier and more enjoyable. Therefore, once the initial build had been completed, it was a priority to present it to some users, ask them to try it out for themselves and then hear what they thought of it. There were several aspects of the application about which their views were sought:

- Ease of download and installation
- Visual integration with IntelliJ
- How intuitive it was to learn.
- Usefulness of its error reporting
- Similarity of its syntax to course notation.
- Its coverage of Java constructs used on the course.
- When and how it could be used, such as in lectures, tutorials or individual study
- Ideas for further development.
- Any comments or particular likes or dislikes.

A usability questionnaire (Figures 5.1 and 5.2) was designed to collect information on these points. The questionnaire does not contain ranking questions, as respondents often just tick the highest-score boxes for these to avoid appearing too critical. Instead, it uses open-ended questions to encourage candour. I also wrote a User Guide to accompany the evaluation. These items were made available to users before the evaluation, along with the plugin itself.

A presentation was made to students, using course tutorial questions to demonstrate the syntax and capabilities of the plugin. There were also additional exercises, including a past exam question, for students to attempt by themselves.

The presentation was made the week before the students' year-end exams, so understandably they did not have much time to spend on the evaluation. Anticipating this, I also made a presentation to some of my Fourth Year course mates, distributed the assessment materials to them and asked for their feedback.



Veri-J Usability Questionnaire

Thank you very much for taking part in the Veri-J demo session and for providing your feedback about Veri-J. Veri-J is being developed to provide a practical complement to the theory-based learning on the course. It will get used only if students find that it makes learning easier and more enjoyable, so your feedback really is indispensable to its success.

Please do participate in this evaluation if you can. I appreciate that you may not have had time before the demo session to download and install the software or read the user guide. That doesn't matter at all - first impressions are as useful as detailed analysis.

Hard copies of this questionnaire will be distributed at the demo session; please hand them in before you leave. For those unable to attend the demo, the questionnaire is also available as a Word document on the course repo and can be emailed to me at rpk17@ic.ac.uk.

1. How familiar are you with Veri-J so far?

   I have downloaded and installed Veri-J Y/~~N~~

   I have read the Veri-J user guide Y/~~N~~

   I have used Veri-J by myself Y/~~N~~

   I have attended ~~the~~ a demo session. Y/~~N~~

2. Which operating system (with version) are you using Veri-J on?

   WINDOWS 10 PROFESSIONAL.

3. What was your experience of downloading and installing Veri-J? Were the instructions clear? Did everything work first time?

   IT INSTALLED FIRST TIME WITHOUT PROBLEMS. (I SUGGEST THE USER GUIDE COULD ALSO MENTION INSTALLING VIA INTELLIJ CUSTOM REPO - LIVE UPDATE).

4. How consistent do you think the 'look and feel' of Veri-J is with the rest of IntelliJ? Please comment with reference to the main IntelliJ window and any other IntelliJ tool windows you have used such as Debug, Git or Gradle.

   SAME AS REST OF INTELLIJ.

5. How intuitive did you find Veri-J's tool window to use? Please tick one:

   I guessed how most/all elements of the Veri-J tool window worked without reading the user guide. ____

   I needed to refer to the user guide but then fully understood how Veri-J worked. ✓

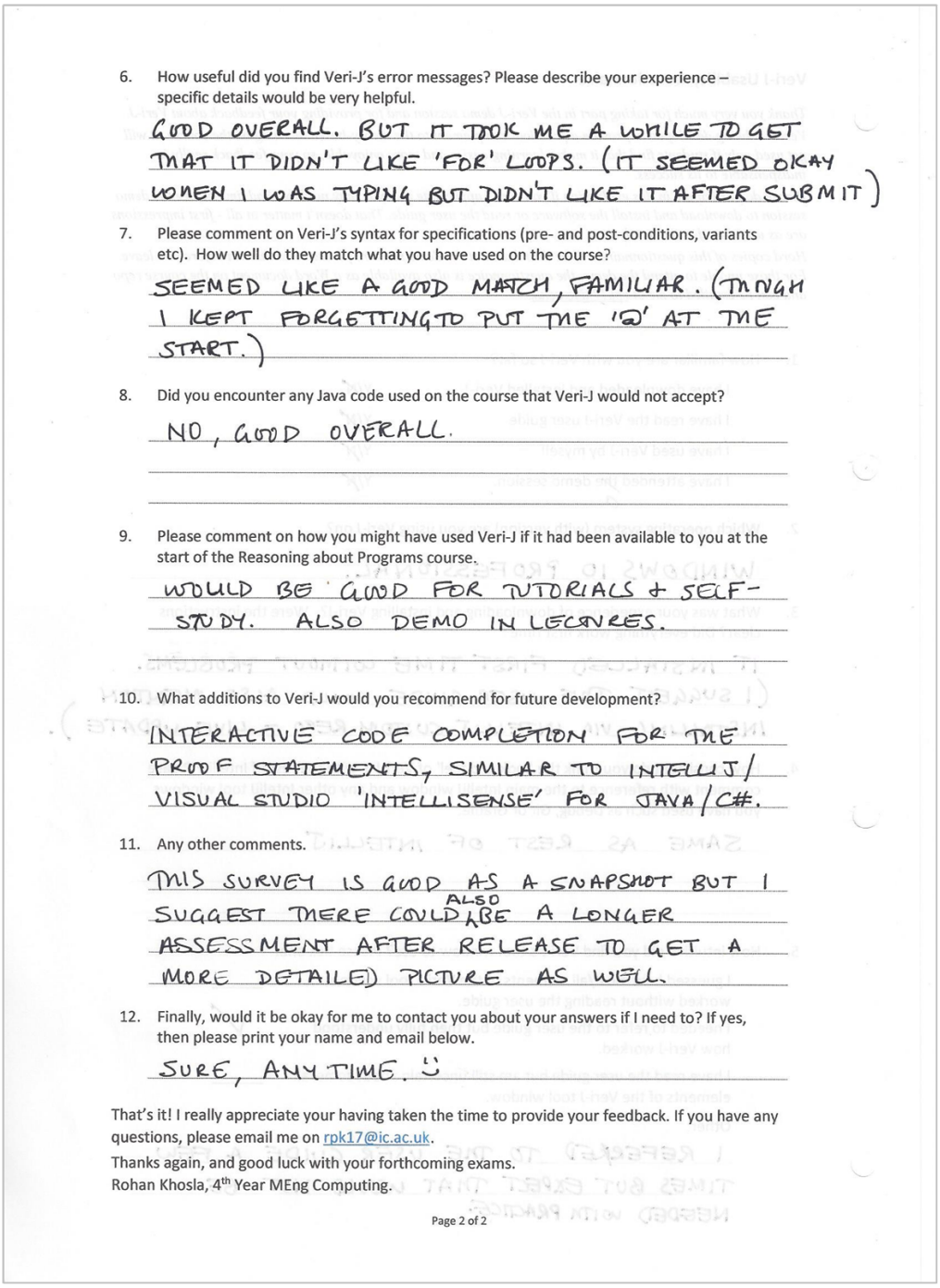
   I have read the user guide but am still uncertain about some/all elements of the Veri-J tool window. ____

   Other:

   I REFERRED TO THE USER GUIDE A FEW TIMES BUT EXPECT THAT WOULD NOT BE NEEDED WITH PRACTICE.

Page 1 of 2

Figure 5.1: Usability questionnaire Page 1 - sample response



6. How useful did you find Veri-J's error messages? Please describe your experience – specific details would be very helpful.

GOOD OVERALL. BUT IT TOOK ME A WHILE TO GET THAT IT DIDN'T LIKE 'FOR' LOOPS. (IT SEEMED OKAY WHEN I WAS TYPING BUT DIDN'T LIKE IT AFTER SUBMIT)

7. Please comment on Veri-J's syntax for specifications (pre- and post-conditions, variants etc). How well do they match what you have used on the course?

SEEMED LIKE A GOOD MATCH, FAMILIAR. (THOUGH I KEPT FORGETTING TO PUT THE '@' AT THE START.)

8. Did you encounter any Java code used on the course that Veri-J would not accept?

NO, GOOD OVERALL.

9. Please comment on how you might have used Veri-J if it had been available to you at the start of the Reasoning about Programs course.

WOULD BE GOOD FOR TUTORIALS & SELF-STUDY. ALSO DEMO IN LECTURES.

10. What additions to Veri-J would you recommend for future development?

INTERACTIVE CODE COMPLETION FOR THE PROOF STATEMENTS, SIMILAR TO INTELLIJ/VISUAL STUDIO 'INTELLISENSE' FOR JAVA/C#.

11. Any other comments.

THIS SURVEY IS GOOD AS A SNAPSHOT BUT I SUGGEST THERE COULD ALSO BE A LONGER ASSESSMENT AFTER RELEASE TO GET A MORE DETAILED PICTURE AS WELL.

12. Finally, would it be okay for me to contact you about your answers if I need to? If yes, then please print your name and email below.

SURE, ANYTIME. :)

That's it! I really appreciate your having taken the time to provide your feedback. If you have any questions, please email me on rpk17@ic.ac.uk.
Thanks again, and good luck with your forthcoming exams.
Rohan Khosla, 4th Year MEng Computing.

Page 2 of 2

Figure 5.2: Usability questionnaire Page 2 - sample response

5.1.2 Results and changes made

The plugin was positively received. The key points in feedback and the actions I have taken in response to them are described below.

Installation

This was trouble free, though it was noted that using IntelliJ's 'Custom repository' feature would enable the plugin to be kept up-to-date without user intervention. In response, I have updated the

user guide to include use of a custom repo as the preferred way of installation.

Appearance

The look and feel of the plugin was considered to be the same as other IntelliJ tool windows and components.

Intuitiveness

Users needed to refer to the user guide at the start but expected to need it less as they became more familiar with the application.

Error reporting

Users found this worked correctly. However, the error message when a `for` loop was entered was found to be confusing. The `for` loop is a feature of Java that is not included in Veri J's translation. As described in section 4.4.1, the rules for such constructs have been left in the grammar but they do not have a visitor method in the `Visitor` class. Therefore, these constructs do not raise either syntax or semantic errors. Instead, Veri-J simply omits them from the translation and relies on Dafny to report a verification error.

In principle, I could have added a visitor method to return an error for each Java construct not included in the translation, which would forestall their being submitted to Dafny. I decided not to do this during the build because the large number of such methods would obscure the methods that are actually needed for the translation.

However, user feedback suggests that the for loop should be an exception to this rule; it is perhaps inevitable that users exploring the plugin will try entering a for loop out of curiosity. I have therefore added a visitor method for the for statement to the `Visitor` class. This traps usage of the `for` loop and reports an error if it used.

New syntax

Veri-J's syntax was judged to be familiar, close to that used on the course. However, use of the at sign @ in the start tag did not come naturally. The need for this has been highlighted in the user guide.

Code coverage

No language constructs required on the course but not included in Veri-J were found.

Acceptance and potential scope

Users felt that Veri-J could be used to support learning on any part of the course, during both contact hours and self-study.

Future development

There were two suggestions for further development:

- Interactive code completion for verification statements. I had already considered including such 'IntelliSense' features when planning the implementation but did not do so as it would have been impracticable within the time available.
- Object Oriented programming features. These were not included in Veri-J as they are not currently relevant to the course.

Code completion is discussed in the final chapter of this report as a possible extension to the plugin in future.

Other comments

Users made the point that while a snapshot evaluation like the one conducted is useful, it should be repeated over a longer period when the application is rolled out for production. I have included this recommendation in the final chapter of this report.

5.2 Performance assessment

5.2.1 Objectives

The objectives of the performance assessment were firstly to determine whether the plugin would perform acceptably in use and, secondly, to identify areas for performance improvement if required in future.

5.2.2 Initial considerations

Whenever Veri-J displays a verification result, it also reports how long each of its major components has taken to process the input. This shows that for repeated attempts with the same input, the total time declines sharply for approximately the first five attempts and then more gradually to reach a steady state approximately after the tenth.

This change in responsiveness could affect users' experience of the tool, so the performance evaluation needed to explore it further. As caching is usually a significant contributor to performance improvement in many applications, I began by exploring how it works in ANTLR and Dafny.

5.2.3 The effect of caching on performance

ANTLR speeds up its top-down parsing algorithm by spawning multiple sub-parsers at each node instead of trying to match sub-nodes one at a time. After parse completion, the parser caches a record of this traversal and reuses it to speed up the next parse. As a further refinement, the cache also includes a record of the successful pathway, which is used to prioritise spawning of the sub-parsers in subsequent parses [37] [38]. As a result:

- Parsing time for all input files progressively reduces as the parser covers more and more of the grammar in its cache.
- Repeated parsing of the same file shows a particular decrease, as the parser responds by increasing the weighting given to the successful path each time.

Dafny also uses a cache to improve performance. Users usually develop verifiable specifications incrementally, through a series of interactions with the program verifier. Dafny caches the program's control-flow graph each time. At each attempt, it uses the cache to identify the parts of

the program affected by the most recent changes and improves its responsiveness by submitting only those parts for re-verification. Published tests demonstrate over a ten-fold improvement in performance as a result [39].

In both cases, the use of caching could explain the performance improvements reported by Veri-J.

5.2.4 Methodology

The discussion on caching suggests that a meaningful picture of Veri-J's response times in use could be built up by repeatedly processing a variety of different input files and measuring start, warm-up and steady state response times for each of the different stages of Veri J's execution. These stages are:

- setup: obtaining a reference to the Java input file and initializing start-up variables;
- translation: running the lexer, parser and visitor on the input file, including semantic checks and Dafny-to-Java mapping.
- verification: running the Dafny CLI on the translator output and reporting the feedback to the user.
- Verification error lookup: using the error map with error coordinates reported by Dafny to obtain the location of verification errors in the Java input file.

The test used a sample of eight methods of varying difficulty, chosen from course tutorials and the course tutors' repo to include all features supported by Veri-J (see Appendix A). All methods were annotated with proof statements. The test was conducted in two parts.

For the first part, the input files had been pre-verified, so the time for error verification lookup was always zero. Each file was verified sixty times. Times for the first and fifth iterations were recorded, along with the mean time from the eleventh to the sixtieth iterations. These represented the start, warm-up and steady state response times respectively and are shown in Tables 5.1 to 5.3.

In the second part, a key element of the specification such as an invariant or post-condition was removed from each input file so that it would no longer verify. The verification error lookup time could then be measured. However, I had observed in use that although the error lookup time reported in use would be non-zero at the first attempt, it would drop to zero or close to zero for the second or subsequent attempts with the same error[1]. Therefore, calculating the mean error lookup time for repeated attempts would be misleading. Instead, to provide realistic portrayal of user experience, this part of the test recorded the times for a single verification attempt only. The results are shown in Table 5.4.

The tests were run on a 64-bit Windows 10 machine having a 2.50GHz Intel i5-7200U processor and 16.0 GB of RAM.

[1]The cause of this behaviour may be that the JVM caches the error location map.

Method identifier	Duration of each stage (ms)			Total Time (ms)
	Setup	Translation	Verification	
triple	1 0.02%	4 0.14%	3308 99.84%	3313
dodgeyDouble	1 0.01%	8 0.38%	2135 99.61%	2144
product	1 0.01%	29 1.16%	2525 98.82%	2556
eqNo	1 0.00%	29 1.11%	2622 99.89%	2652
concat	1 0.01%	38 1.31%	2901 98.68%	2940
calcThreeProduct	1 0.01%	15 0.71%	2157 99.28%	2172
isAlmostPerfectNumber	1 0.02%	11 0.41%	2885 99.57%	2897
substring	1 0.01%	45 1.25%	3559 98.74%	3604
mean	0.01%	0.81%	99.18%	2785

Table 5.1: Start execution times (first attempt)

Method identifier	Duration of each stage (ms)			Total Time (ms)
	Setup	Translation	Verification	
triple	1 0.02%	3 0.16%	2409 99.82%	2052
dodgeyDouble	0 0.01%	6 0.29%	2092 99.70%	2098
product	1 0.02%	4 0.17%	2488 99.81%	2493
eqNo	1 0.02%	30 1.20%	2528 98.78%	2559
concat	1 0.01%	41 1.45%	2798 98.54%	2840
calcThreeProduct	1 0.01%	17 0.83%	2112 99.15%	2130
isAlmostPerfectNumber	1 0.01%	11 0.44%	2653 99.55%	2665
substring	1 0.01%	33 1.15%	2921 98.85%	2955
mean	0.01%	0.71%	99.28%	2474

Table 5.2: Warm-up execution times (fifth attempt)

Method identifier	Duration of each stage (ms)			Total Time (ms)
	Setup	Translation	Verification	
triple	1 0.02%	4 0.20%	2020 99.78%	2025
dodgeyDouble	1 0.02%	6 0.32%	2022 99.66%	2029
product	0 0.01%	4 0.17%	2461 99.82%	2465
eqNo	1 0.01%	28 1.13%	2526 98.86%	2555
concat	1 0.01%	37 1.33%	2790 98.66%	2828
calcThreeProduct	1 0.01%	17 0.81%	2095 99.18%	2112
isAlmostPerfectNumber	0 0.01%	13 0.51%	2558 99.48%	2571
substring	1 0.02%	31 1.09%	2894 98.89%	2926
mean	0.01%	0.70%	99.29%	2439

Table 5.3: Steady state execution times (mean of 11th to 60th attempts)

Method identifier	Duration of each stage (ms)				Total Time (ms)
	Setup	Translation	Verification	Error Mapping	
triple	1 0.01%	8 0.38%	2177 99.61%	0 0.00%	2186
dodgeyDouble	0 0.02%	28 1.20%	2303 98.78%	0 0.00%	2331
product	0 0.00%	28 0.99%	2844 99.00%	2 0.07%	2186
eqNo	0 0.01%	39 1.47%	2629 98.52%	0 0.02%	2669
concat	0 0.01%	39 1.44%	2706 98.55%	3 0.12%	2749
calcThreeProduct	0 0.02%	13 0.61%	2186 99.36%	4 0.18%	2204
isAlmostPerfect Number	0 0.01%	50 1.84%	2709 98.15%	3 0.12%	2764
substring	0 0.03%	33 1.13%	2925 98.85%	2 0.07%	2961
mean	0.01%	1.13%	98.79%	0.07%	2592

Table 5.4: Execution times with verification error (single attempt)

5.2.5 Interpretation

Veri-J took an average of between two and three seconds to verify the methods in the test. This is a fast response time and means that users will be able to run the plugin repeatedly without being concerned about having to wait for a result each time.

The response is dominated by the Dafny/Z3 solver which, on average, accounts for 99.14% of the total time taken. By contrast, the translator takes up merely 0.74% of the time on average during a successful verification. Therefore, optimising the translator would be of negligible benefit to the user and improving the performance of the plugin meaningfully beyond its current level would require improvements to Dafny or Z3.

Table 5.4 shows that verification error lookup took 4 ms or less, accounting for 0.07% of total time on average. Again, this is fast and should not be an issue for users as they work their way progressively towards a solution to a problem.

Finally, we can see that Tables 5.1 to 5.3 show that the verification time for all methods declines from start to warm up to steady state, which is as expected from the earlier discussion on caching. However, the corresponding translation times do not display a similar decrease. This may be because these times are so short that any patterns in the data are masked by environmental noise.

5.3 Reflection

The evaluation shows that the plugin has the features and responsiveness it needs to be useful to students on the course. Its advantages are:

- Coverage: Veri-J covers all of the Java and specification constructs used on the course, in the same notation. Students will be able to familiarise themselves with it quickly and begin using it from the start of the course.
- Error messaging: Veri-J's interactive syntax checker displays helpful messages as input is being typed. Semantic checks prevent errors such as incorrect use of reserved words. Token-level verification error mapping ensures that the location of verification errors, including those in imported files, is precisely reported and highlighted.
- A gateway to Dafny: Veri-J displays the Dafny translation prominently in its UI and provides direct access for this to be explored in Visual Studio Code.

The evaluation also identified interactive code completion of verification statements as a useful extension for the plugin in future. A further recommendation was that the evaluation should be repeated over a longer period as part of the plugin's rollout to users. Both of these points are discussed in the final chapter of this report.

Chapter 6

Ethics

6.1 Short-term considerations

The immediate ethical considerations when using Veri-J arise from the risk of its producing false positives or negatives. As students may use it extensively to help with their coursework and learning, incorrect responses could cost them marks.

A more insidious risk arises from occasional mismatches between the approach taken by the Z3 SAT/SMT solver and that required on the course. Sometimes, proof statements required for solutions in the Reasoning sub-module are not required by Z3 to prove correctness. Students who leave these out of their work could lose marks. Conversely, a proof that is sufficient on the course may be unprovable by Z3 without additional lemmas and statements, leading to delays and frustration.

One measure to mitigate these risks is to have a period of ongoing evaluation and support when the plugin is introduced each year, as identified during the user evaluation. A member of the DOC team could be nominated as the help contact for the duration of the module. Another measure, already taken, is to alert users to these issues in the user guide, with the advice not to rely solely on the plugin for their work.

6.2 Longer-term considerations

Though use of Dafny is widespread, this is mainly as a tool for teaching and research. Because of the limitations of current technology, Dafny's usefulness at present is more as a means to improve software engineering skills than as a software engineering tool in its own right.

However, it is reasonable to predict that this will change as technology moves forward. For example, Z3 may move from the sixty-year-old DPLL algorithm to CDCL, which could increase its performance considerably. With such improvements, it may become practicable to test large programs as a matter of course. It is credible to think that even AI systems, whose internal logic today often performs as a fallible black box, could someday routinely be proven to be predictable and trustworthy.

This matters because, currently, the debate on the ethics of the use of many software applications hinges on the risks of their going wrong. For example, air regulators currently ban AI autopilots because their responses cannot be guaranteed. And public opinion is mostly against autonomous weapons in case they target civilians. If all such errors could be eliminated with certainty, the debate would shift to the next question: does the fact that an action is always correct justify allowing it to take place?

For example, is it morally acceptable to allow a conscienceless drone the autonomy to take human life, even though it only ever kills the enemy? More mundanely, what effect will checkout-less stores have on society? Many people rely on their shopping trips for social contact. A recent online petition for a superstore chain to provide more manned checkouts attracted over 110,000 signatures[40]. Does increased profit and convenience for some justify increased isolation for others?

The development of increasingly error-free software is pushing such questions to the forefront of public discussion about the introduction of new technology. My purpose in referring to these debates here is not to engage in them but rather to note that progress even in a specialist area like formal methods can eventually affect every corner of society. Veri-J is, perhaps, a very small step in that direction.

Chapter 7

Conclusion

7.1 Achievements

Veri-J is an interactive verification tool that includes all the Java and proof constructs used on the Reasoning sub-module. It provides a practical, hands-on element to the course and allows students to experiment and use feedback to build up proofs step-by-step.

The plugin syntax for proof statements and expressions closely matches the notation already used on the course. This includes pre-, post- and mid-conditions, variants and invariants. It also supports array slices, entry-state values, quantifiers and imports.

Error reporting is supported for syntax, semantic and verification errors. Syntax error checking occurs interactively with each keystroke. Semantic checks prevent use of reserved words as identifiers and of many unsupported constructs. Verification error reporting is enabled by a mapping of Dafny to Java locations, which includes locations in imported files.

The plugin has an extensible, modular structure, which will allow it to be used with IDEs other than IntelliJ or as a command-line instruction with little modification. Also, it uses a complete Java grammar, so the addition of further Java constructs to the translator would be straightforward.

7.2 Further work

Counterexamples

As described in section 2.1, a counterexample is a set of values of parameters and variables that demonstrates that a proof is invalid. Knowing these values helps users to understand why a proof has not verified. Counterexamples are passed to Dafny by Z3 but currently are available only in the Dafny extension for Visual Studio Code and not via the Dafny CLI. However, Dafny's GitHub site[22] states that this feature is planned for a future release of the CLI. When that happens, it would be useful for Veri-J's `Verifier` class to be modified to display the counterexample in the feedback panel.

Code completion

Interactive code completion is already available in IntelliJ for Java code. A similar feature could be included for verification statements and methods as well. Starting with the '//@' tag, popup lists would appear progressively to help the user complete the statement. This has not been included

in the current version of the plugin due to time constraints. However, feedback from users during the evaluation suggests it would be useful to do so in future.

Embedded inc/dec expressions and method calls

As described in section 4.5.9, the plugin does not allow increment/decrement expressions and method calls to be embedded in other expressions. Although the plugin offers a workaround, it would be preferable if it could allow their use. The principles of implementing this are outlined in section 4.5.9 and the annotations to the grammar needed for it have already been made.

'for' loops and 'switch' statements

for loops and switch statements are not currently used in the Reasoning sub-module. Because of this, and also due to time constraints on the project, Veri-J returns an error message preventing their use. However, if these constructs are needed on the course in future, extending the code to include them should be straightforward as the visitor methods for them are already in place. Furthermore, both have equivalent statements in Dafny. The Java for loop could be translated to either a for loop, a forall loop or a while loop in Dafny. The Java switch statement is similar to Dafny's matches statement or it could be translated to an if...else statement in Dafny.

Abstract syntax tree

At the start of this project I considered using an abstract syntax tree (AST) to generate the Dafny translation rather generating it directly from the parse tree visitor. I did not do so because I felt that the project did not justify the complexity that an AST would have introduced. However, an AST would be useful if the project is extended substantially at some point, for example by the inclusion of OO features. This might be something that could be addressed in future.

Further usability evaluation

Finally, the usability evaluation described in Chapter 5 provided useful feedback but was limited by the time available. It would be useful to conduct a similar survey over a longer period of time when the plugin is released to students, to understand how it is being used and to provide the support required for rollout.

Appendix A

Evaluation Code

A.1 triple method

```
public class tut_wk07_q01_triple {

    int triple(int x)
    //@ POST : r == 3 * x;
    {
        int y = x;
        //@ MID: y == x;
        y = y + x;
        //@ MID: y == 2 * x;
        y = y + x;
        //@ MID: y == 3 * x;
        return y;
    }
}
```

A.2 dodgeyDouble method

```
public class tut_wk07_q02_dodgeyDouble {

    int dodgeyDouble (int x)
    //@ PRE: true;
    //@ POST: r == 2 * Pre(x);
    {
        //@ MID: x == Pre(x);
        int y = x;
        //@ MID: y == x && x == Pre(x);
        x = x + 1;
        //@ MID: y == Pre(x) && x == Pre(x) + 1;
        y = y + y;
        //@ MID: y == 2 * Pre(x);
        return y;
    }
}
```

A.3 product method

```
public class tut_wk09_q05_prod {

    int product (int[] a)
    //@ PRE: a != null;
    //@ PRE: 0 < a.length; //Extra precondition needed by Dafny.
    //@ POST: r == Pi a[0..a.length);
    {
        //Start from a[1] not a[0], otherwise the non-inclusive
        //upperbound of the array slice in the invariant will
        //be -1 at the start.
        int res = a[0];
        int i = 1;

        //@ INV: 0 <= i <= a.length && res == Pi a[..i);
        //@ VAR: a.length - i;
        while (i < a.length) {
            res = res * a[i];
            ++i;
        }
        //@ MID: res == Pi a[..);
        return res;
    }
}
```

A.4 eqNo method

Method `eqNo` counts the number of equal (i.e., same position and value) elements in two integer arrays of equal length.

```
public class tut_wk09_q06_eqNo {

    //@FUNC
    public static int eqsAux(int[] a, int[] b, int i)
    //@ PRE: a != null && b != null && a.length == b.length;
    //@ PRE: 0 <= i <= a.Length;
    //@ VAR: a.length - i ;
    {
        return (i == a.length)?
                    0
                :
                    (a[i] == b[i])?
                            eqsAux(a, b, i + 1) + 1
                    :
                            eqsAux(a, b, i + 1);
    }


    int eqNo (int[] a, int[] b)
    //@ PRE: a != null && b != null && a.length == b.length;
    //@ POST: r == eqsAux(Pre(a[..]), Pre(b[..]), 0);
    {

        //@ MID: a != null && b != null && a.length == b.length
            && a[..] == Pre(a[..]) && b[..] == Pre(b[..]);

        int res = 0, i = a.length;

        //@ INV: a[..] == Pre(a[..]) && b[..] == Pre(b[..]) && 0
             <= i <= a.length && res == eqsAux(a[..], b[..], i);
        //@ VAR: i;
        while (i > 0) {
            i--;
            if(a[i] == b[i]) {res++;}
        }
        //@ MID: a[..] == Pre(a[..]) && b[..] == Pre(b[..]) &&
            res == eqsAux(a[..], b[..], 0);
        return res;
    }
}
```

A.5 concat method

Method `concat` constructs the concatenation of two input strings, denoted by `a[..) : b[..)`

```
public class tut_wk10_q01_arrayConcat {

    char[] concat(char[] a, char[] b)
    //@ PRE: a != null && b != null;
    //@ POST: a[..) == Pre(a[..)) && b[..) == Pre(b[..)) && r
        [..) == a[..) : b[..);
    {

        //@ MID: a != null && b!= null && a[..) == Pre(a[..))
            && b[..) == Pre(b[..));

        int x = a.length + b.length;
        //@ MID: x == a.length + b.length;
        char[] c = new char[x];
        //@ MID: c != null && c.length == a.length + b.length;
        copy(c, a, 0);
        //@ MID: c[..a.length) == a[..);
        copy(c, b, a.length);
        //@ MID: c[..) == a[..) : b[..);

        return c;
    }

    //Function concat makes use of an auxiliary function copy,
        implemented as follows:
    void copy (char[] a, char[] b, int x)
    //@ PRE: a != null && b != null && 0 <= x <= a.length -
        b.length;
    //@ POST: b[..) == Pre(b[..)) && a[..) == Pre(a[..x))
        : b[..) : Pre(a[x + b.length ..));
    //@ MODIFIES: a;
    {

        //@ MID: a != null && b!= null && a[..) == Pre(a[..)) &&
            b[..) == Pre(b[..)) && 0 <= x <= a.length - b.length;

        int i = 0;

        //@ INV: a != null && b != null && 0 <= x <= a.length -
            b.length && b[..) == Pre(b[..)) && 0 <= i
            <= b.length && INV: a[..) == Pre(a[..x)) : b[..i)
            : Pre(a[x + i ..));
        //@ VAR: b.length - i;
        while (i < b.length)
        {
            a[x+i] = b[i];
            i++;
        }
    }
}
```

A.6 calcThreeProduct method

Method `calcThreeProduct` calculates `m * n * p` using only increments and decrements of 1 and comparisons with 0.

```
public class tut_wk11_q04_triplyNestedLoops_mxnxp {

    int calcThreeProduct(int m, int n, int p)
    //@ PRE: m >= 0 && n >= 0 && p >= 0;
    //@ POST: r == m * n * p;
    {

        int m1 = 0;
        int res = 0;

        //@ INV: res == m1 * n * p && m1 <= m;
        //@ VAR: m - m1;
        while (m1 != m) {
            int n1 = 0;

            //@ INV: res == (m1 * n * p) + (n1 * p) && m1 < m &&
                 n1 <= n;
            //@ VAR: n - n1;

                // V2
            while (n1 != n) {
                int p1 = 0;

                //@ INV: res == (m1 * n * p) + (n1 * p) + p1 &&
                    m1 < m && n1 < n && p1 <= p;       // I3
                //@ VAR: p - p1;

                    // V3
                while (p1 != p) {
                    res = res + 1;
                    p1 = p1 + 1;
                }
                //@ MID: res == (m1 * n * p) + (n1 + 1) * p &&
                    m1 < m && n1 < n;          // M1
                n1 = n1 + 1;
            }
            //@ MID: res == (m1 + 1) * n * p && m1 < m;
                                                  // M2
            m1 = m1 + 1;
        }
        //@ MID: res == m * n * p;

            // M3
        return res;
    }
}
```

A.7 almostPerfectNumber method

An 'almost perfect number' is a positive integer that is equal to one more than the sum of its positive divisors, excluding the number itself. e.g. $8 = 1 + (1 + 2 + 4)$. Adapted from code posted on the Reasoning sub-module tutors' repo.

```
package testFiles;

//import static testFiles.Eval_aux.*;
import static testFiles.Eval_aux.auxAlmostPerfectSum;
import static testFiles.Eval_aux.almostPerfectNumber;

public class Eval_AlmostPerfectNumber {

    boolean isAlmostPerfectNumber(int n)
    //@ PRE: n > 0;
    //@ POST: r == almostPerfectNumber(n);
    {
        int i = 1, rez = 0;

        //@ INV: 1 <= i <= n && rez == auxAlmostPerfectSum(i -
            1, n) && n == Pre(n);
        while(i < n)
        {
            if (n % i == 0)
            {
                rez += i;
            }
            i++;
        }

        //@ MID: rez == auxAlmostPerfectSum(n - 1, n) &&
            n == Pre(n);

        return rez == n - 1;
    }
}


package testFiles;

public class Eval_aux {

    //@ FUNC
    public static int auxAlmostPerfectSum(int i, int n)
    //@ PRE: 0 <= i < n;
    {
        return (i != 0) ? ((n % i == 0) ? (i +
            auxAlmostPerfectSum(i - 1, n)) : (auxAlmostPerfectSum
            (i - 1, n))) : 0;
    }

    //@ FUNC
    public static boolean almostPerfectNumber(int n)
    //@ PRE: n > 0;
    {
        return n - 1 == auxAlmostPerfectSum(n - 1, n);
    }
}
```

A.8 substring method

Method `subString` returns the position of one string in another, if found. Adapted from code posted on the Reasoning sub-module tutors' repo.

```
public class Eval_Substring {


    boolean MatchedAll(char[] s, char[] t, int i)
    //@ PRE: s != null && t != null;
    //@ PRE: 0<= i <= s.length - t.length;
    //@ POST: r <==> Match(s, t, i, t.length);
    {
        int j = 0;

        //@ INV: 0 <= j <= t.length && Match(s, t, i, j);
        while(j < t.length && s[i + j] == t[j])
        {
            j++;
        }

        //@ MID: j != t.Length ==> s[i+j] != t[j];
        // not necessary, but speeds up the theorem prover

        return j == t.length;
    }

    int substring(char[] s, char[] t)
    //@ PRE: s != null && t != null;
    //@ PRE: s.length > t.Length ;
    //@ POST:  0<= r <= s.Length;
    //@ POST:  r == s.length ==> forall k :: 0 <= k < s.length -
         t.length && LookHere(k) ==> !Match(s, t, k, t.length);
    //@ POST:  r < s.length  ==>  Match(s, t, r, t.length) &&
        forall k :: 0 <= k < r && LookHere(k) ==> !Match(s, t, k,
         t.length);
    {

        int i = 0, j = 0;
        boolean found =  false; // t.Length == j; (*)

        //@ INV: found <==> Match(s, t, i - 1, t.Length);
        //@ INV: forall k :: 0 <= k < i-1 && LookHere(k) ==> !
            Match(s, t, k, t.Length);
        //@ INV: 0 <= i <= s.Length;
        while(i < s.length - t.length && !found)
        {
            found = MatchedAll(s,t,i);
            i++;
        }

        if (found)
        {
            return i - 1;
        }
        else {
            return s.length;
        }
    }
```

```
    //@ PRED
    public static void LookHere(int k)
    {
        //@ true;
    }

    //@ PRED
    public static void Match(char[] s, char[] t, int m, int n)
    //@ PRE: s != null && t != null;
    {
        //@ 0 <= m && 0 <= m + n <= s.length && 0 <= n <= t.
            length && (forall k :: 0 <= k < n  ==> s[m + k] == t[
            k]);
    }
}
```

Appendix B

Further examples

B.1 BubbleSort

Method `bubbleSort` takes an integer array as its in-parameter and contains two nested loops. The outer loop iterates from the end of the array towards the start. The inner loop iterates from the start of the array towards the current index of the outer loop, swapping out-of-order pairs at each step to propagate the largest element towards the end of the array. This creates a sorted partition in the array which starts at its end and grows by one place towards the beginning with each iteration of the outer loop. Adapted from a lecture at Carnegie Mellon University[41]

```
public class BubbleSort {

    void bubbleSort (int[] a)
    //@ PRE: a != null;
    //@ POST: sorted (a , 0 , a.length -1);
    //@ MODIFIES: a;
    {
        int i = a.length - 1;

        //@ INV: i < 0 ==> a.length == 0;
        //@ INV: -1 <= i < a . length;
        //@ INV: sorted (a , i , a . length -1);
        //@ INV: partitioned (a , i );
        while ( i > 0)
        {
            int j = 0, t;

            //@ INV: 0 <= j <= i < a.length; //&& 0 <= j <= i;
            //@ INV: sorted (a , i , a.length -1);
            //@ INV: partitioned (a, i);
            //@ INV: forall k :: 0 <= k <= j ==> a [k] <= a [j];
            while ( j < i )
            {
                if ( a [j] > a [j +1])
                {
                    t = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = t;
                }
                j++;
            }
            i--;
        }
    }
```

```
    //@ PRED
    public static void sorted ( int[] a, int lo, int hi)
    //@ PRE: a != null;
    {
        //@ forall i , j :: 0 <= lo <= i <= j <= hi < a.length
            ==> a [i] <= a [j];
    }

    //@ PRED
    public static void partitioned (int[] a, int i)
    //@ PRE: a != null;
    {
        //@ forall m, n :: 0 <= m <= i < n < a.length ==> a [m]
            <= a [n];
    }
}
```

B.2 SelectionSort

Method `selectionSort` with helper method `findMin` implements a selection sort. Adapted from a discussion on Stack Overflow[42].

```
public class SelectionSort {

    //@ PRED
    public static void sorted(int[] a, int i)
    //@ PRE: a != null;
    //@ PRE: 0 <= i <= a.length;
    {
        //@ forall k :: 0 < k < i ==> a[k-1] <= a[k];
    }

    int findMin(int[] a, int i)
    //@ PRE: a != null;
    //@ PRE: 0 <= i < a.length;
    //@ POST: i <= r < a.length;
    //@ POST: forall k :: i <= k < a.length ==> a[k] >= a[r];
    {
        int j = i ;
        int res = i;

        //@ INV: i <= j <= a.length;
        //@ INV: i <= res < a.length;
        //@ INV: forall k :: i <= k < j ==> a[k] >= a[res];
        //@ VAR: a.length - j; //not required
        while(j < a.length)
        {
            if(a[j] < a[res]){res = j;}
            j++;
        }
        return res;
    }

    void selectionSort(int[] a)
    //@ PRE: a != null;
    //@ POST: a != null;
    //@ POST: sorted(a,a.length);
    //@ MODIFIES: a;
    {
        int c = 0;
        int t;

        //@ INV: 0 <= c <= a.length;
        //@ INV: sorted(a,c);
        //@ INV: forall k, l :: 0 <= k < c <= l < a.length ==> a
            [k] <= a[l];
        //@ VAR: a.length - c; //not required
        while(c < a.length)
        {
            int m = findMin(a,c);

            t = a[c];
            a[c] = a[m];
            a[m] = t;

            c++;
        }
    }
}
```

B.3 SkippingLemma

Method `findZero` returns the index of the first zero in an integer array having two properties: (a) all elements are whole numbers and (b) each element is either equal to or one less than the element before it. This means that `findZero` can skip from position `j` to position `j + a[j]` without having to check anything in between. However, although Dafny/Z3 is told the relationship between consecutive elements in a precondition, it cannot use this to infer the relationship between elements that are further apart. It needs the help of `skippingLemma` to do this. Adapted from a discussion on the Dafny Git repo[43].

```
public class SkippingLemma {

    //@ LEMMA
    public static void skippingLemma(int[] a, int j)
    //@ PRE: forall i :: 0 <= i < a.length ==> 0 <= a[i];
    //@ PRE: forall i :: 0 < i < a.length ==> a[i-1]-1 <= a[i];
    //@ PRE: 0 <= j < a.length;
    //@ POST: forall k :: j <= k < j + a[j] && k < a.length ==>
        a[k] != 0;
    {
        int i = j;

        //@ INV: i < a.length ==> a[j] - (i-j) <= a[i];
        //@ INV: forall k :: j <= k < i && k < a.length ==> a[k]
             != 0;
        while (i < j + a[j] && i < a.length)
        {
            i ++;
        }
    }

    int findZero(int[] a)
    //@ PRE: forall i :: 0 <= i < a.length ==> 0 <= a[i];
    //@ PRE: forall i :: 0 < i < a.length ==> a[i-1]-1 <= a[i];
    //@ POST: r < 0  ==> forall i :: 0 <= i < a.length ==> a[i]
        != 0;
    //@ POST: 0 <= r ==> r < a.length && a[r] == 0;
    {
        int index = 0;

        //@ INV: 0 <= index;
        //@ INV: forall k :: 0 <= k < index && k < a.Length ==>
            a[k] != 0;
        while (index < a.length)
        {
            if (a[index] == 0) { return index; }
            skippingLemma(a, index);
            index += a[index];
        }
        return -1;
    }
}
```

B.4 FermatTest

Method `fermatTest` applies the Fermat primality test to determine whether a number is a probable prime. Adapted from code posted on the Reasoning sub-module tutors' repo.

```
public class FermatTest {

    //@ FUNC
    public static int pow(int a, int b)
    //@ PRE: b >= 0;
    {
        return (b == 0)? 1 : a * pow(a, b - 1);
    }

    int power(int a, int b)
    //@ PRE: b >= 0;
    //@ POST: r == pow(a, b);
    {
        int i = 0, c = 1;

        //@ INV: 0 <= i <= b && c == pow(a, i);
        while(i < b)
        {
            c *= a;
            i ++;
        }
        return c;
    }

    boolean fermatTest(int p)
    //@ PRE: p >= 1;
    //@ POST:  r ==> (forall i :: 0 < i < p ==> ((pow(i, p - 1)
        % p) == 1));
    {
        boolean test =  true;
        int i = 1;

        //@ INV: 1 <= i <= p && p == Pre(p) && (test ==> (forall
             j :: 0 < j < i ==> ((pow(j, p - 1) % p) == 1)));
        //@ VAR: p - i;
        while(i < p)
        {
            int power = power(i, p - 1);

            if(power % p != 1)
            {
                test = false;
            }
            i++;
        }
        //@ MID: p == Pre(p) && (test ==> (forall j :: 0 < j < p
             ==> ((pow(j, p - 1) % p) == 1)));

        return test;
    }
}
```

B.5 ArraySliceRotator

Method `displace` below rotates an array slice within an array by one place. The `rotated` predicate defines what it means to be rotated: each element in the rotated slice `rot` is the same as the one in the position before it in the original slice `orig`, except for `rot[0]`, which is equal to the last item in orig. Adapted from Dafny code posted on Stack Overflow[44].

```
public class ArraySliceRotator {

    //@PRED
    public static void rotated(int[] orig, int[] rot)
    //@ PRE: o.length == r.length;
    {
        //@ (forall i :: 1 <= i < orig.length ==> rot[i] == orig
            [i-1]) && (orig.length > 0 ==> rot[0] == orig[orig.
            length - 1]);
    }

    // rotates a region of the array by one place forward
    public int[] displace(int[] arr, int start, int len)
    //@ PRE:  arr != null;
    //@ PRE:  start >= 0 && len >=0;
    //@ PRE:  start + len < arr.length;
    //@ POST:  r != null && r.length == arr.length;
    //@ POST: arr[..start) == r[..start);
    //@ POST: arr[(start + len)..) == r[(start + len)..);
    //@ POST: rotated(arr[start .. start+len), r[start .. start+
        len));
    {
        int i = 0;
        int[] res = new int[arr.length];

        //@ INV: i <= start;
        //@ INV: forall int k :: k >= 0 &&  k < i ==> res[k] ==
            arr[k];
        while (i < start)
        {
            res[i] = arr[i];
            i++;
        }

        //@ MID: arr[..start) == res[..start);

        if (len > 0) {

            res[start] = arr[start + len - 1];

            //@ MID: res[start] == arr[start + len - 1];

            i = start + 1;

            //@ INV: start < i <= start + len;
            //@ INV: arr[..start) == res[..start);
            //@ INV: res[start] == arr[start + len - 1];
            //@ INV: forall int k :: start < k < i ==> res[k] ==
                 arr[k-1];
            while (i < start + len)
            {
                res[i] = arr[i - 1];
                i ++;
            }
        }
```

```
        //@MID: rotated(arr[start .. start+len), res[start ..
            start+len));

        i = start + len;
        //@ INV: start + len <= i <= arr.length;
        //@ INV: arr[..start) == res[..start);
        //@ INV: rotated(arr[start .. start + len), res[start ..
             start + len));
        //@ INV: forall int k :: start + len <= k < i ==> res[k]
             == arr[k];
        while (i < arr.length)
        {
            res[i] = arr[i];
            i++;
        }
        return res;
    }
}
```

B.6 ProgressiveSum

Method `sum` takes two arrays `a` and `b` as inputs and modifies `b` so that `b[i] = a[0] + a[1] + ... + a[i]`. Adapted from Dafny code posted on Stack Overflow[45].

```
public class ProgressiveSum {

    //@ FUNC
    public static int sumTo(int[] a, int n)
    //@ PRE: a != null;
    //@ PRE: 0 <= n < a.length;
    //@ VAR: n;
    {
        return (n == 0) ? a[0] : sumTo(a, n-1) + a[n];
    }

//Method sum takes two arrays a and b as inputs and modifies b
//so that b[i] = a[0] + a[1] + ... + a[i].

    public void sum(int[] a, int[] b)
    //@ PRE: a != null && b != null && a != b;
    //@ PRE:  a.length >= 1;
    //@ PRE: a.length == b.length;
    //@ POST: forall x | 0 <= x < b.Length :: b[x] == sumTo(a,x)
        ;
    //@ MODIFIES: b;
    {
        b[0] = a[0];
        int i = 1;

        //@ INV: b[0] == sumTo(a,0);
        //@ INV: 1 <= i <= b.length;
        //@ INV: forall x | 1 <= x < i :: b[x] == sumTo(a,x);
        //@ VAR: b.length - i;
        while (i < b.length)
        {
            b[i] = a[i] + b[i-1];
            i++;
        }
    }
}
```

B.7 Fibonacci

```
public class Fibonacci {

    int ComputeFib(int n)
    //@ PRE: n >= 0;
    //@ POST: r == fib(n);
    {
        if (n == 0)
            return 0;

        int i = 1, a = 0, b = 1, temp;

        //@ INV: 0 < i <= n;
        //@ INV: a == fib(i - 1);
        //@ INV: b == fib(i);
        while (i < n)
        {
            temp = a;
            a = b;
            b += temp;
            i ++;
        }
        return b;
    }

    //@ FUNC
    public static int fib(int n)
    //@ PRE: n >= 0;
    {
        return (n == 0)? 0 : ((n == 1)? 1 : (fib(n - 1) + fib(n
            - 2)));
    }
}
```

B.8 LargestSqrt

Method `sqrt` finds the largest natural number that has a square value that is less than or equal to the given natural number `n`. Adapted from Dafny code posted on Stack Overflow[46].

```
public class LargestSqrt {

    //@ FUNC
    public static int square(int n)
    {
        return n * n;
    }

    int sqrt(int n)
    //@ PRE: n >= 0;
    //@ POST: r * r <= n;
    //@ POST: forall i :: 0 <= i < r ==> square(i) < r * r;
    {
        int i = 0;
        int res = 0;

        //@ INV: res * res <= n;
        //@ INV: forall k :: 0 <= k < res ==> square(k) < res *
            res;
        //@ VAR: n - i;
        while (i * i <= n)
        {
            res = i;
            i++;
        }
        return res;
    }
}
```

B.9 BinarySearch

```
public class BinarySearch {

    int BinarySearch(int[] a, int key)
    //@ PRE: a != null;
    //@ PRE: forall j, k :: 0 <= j < k < a.length ==> a[j] <= a[
        k];
    //@ POST: 0 <= r ==> r < a.length && a[r] == key;
    //@ POST: r < 0 ==> forall k :: 0 <= k < a.length ==> a[k]
        != key;

    {
        int low = 0, high = a.length, index;

        //@ INV: 0 <= low <= high <= a.length;
        //@ INV: forall i :: 0 <= i < a.length && !(low <= i <
            high) ==> a[i] != key;
        while (low < high)
        {
            int mid = (low + high) / 2;
            if (a[mid] < key) {
                low = mid + 1;
            } else if (key < a[mid]) {
                high = mid;
            } else {
                index = mid;
                return index;
            }
        }
        index = -1;
        return index;
    }

}
```

B.10 Find

```
public class Eval_Find {

    int Find(int[] a, int key)
    //@ PRE: a!= null;
    //@ POST: 0 <= r ==> r < a.length && a[r] == key;
    //@ POST: r < 0 ==> forall k :: 0 <= k < a.length ==> a[k]
        != key;
    {
        int index = 0;

        //@ INV: 0 <= index <= a.length;
        //@ INV: forall k :: 0 <= k < index ==> a[k] != key;
        while (index < a.length)
        {
            if (a[index] == key) { return index; }
            index = index + 1;
        }
        index = -1;
        return index;
    }
}
```

B.11 CrudePrime

```
public class Eval_CrudePrime {

    boolean crudePrime(int x)
    //@ PRE: x >= 0;
    //@ POST: r ==> !(exists m :: (x == 2 * m) || (x == 3 * m)
        || (x == 5 * m));
    {
        boolean test = false;
        //@ MID: !test;
        test = (x % 2 == 0);
        //@ MID: !test ==> !(exists m :: (x == 2 * m));
        test = test || (x % 3 == 0);
        //@ MID: !test ==> !(exists m :: (x == 2 * m) || (x == 3
            * m));
        test = test || (x % 5 == 0);
        //@ MID: !test ==> !(exists m :: (x == 2 * m) || (x == 3
            * m) ||(x == 5 * m));
        return !test;
    }

}
```

Bibliography

[1] Wills S. PayPal accidentally credits man $92 quadrillion. 2013 July. Available from: https://edition.cnn.com/2013/07/17/tech/paypal-error/index.html.

[2] Boyer RS, Elspas B, Levitt KN. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. SIGPLAN Not. 1975 apr;10(6):234–245. Available from: https://doi.org/10.1145/390016.808445.

[3] King JC. Symbolic Execution and Program Testing. Commun ACM. 1976 jul;19(7):385–394. Available from: https://doi.org/10.1145/360248.360252.

[4] Jelvis T. Analyzing Programs with Z3; 2016. Available from: https://www.youtube.com/watch?v=ruNFcH-KibY.

[5] Leino KRM, Wilcox J. For an array in Dafny, what's the difference between old(a[0]) and old(a)[0]?; 2020. Available from: https://stackoverflow.com/questions/64628903/for-an-array-in-dafny-whats-the-difference-between-olda0-and-olda0.

[6] Leino KRM. Modifying method parameters; 2018. Available from: https://stackoverflow.com/questions/51172100/modifying-method-parameters.

[7] Leino KRM. Dafny Power User: old and unchanged; 2020. Available from: http://leino.science/papers/krml273.html.

[8] Ma KK, Phang KY, Foster JS, Hicks M. Directed Symbolic Execution. In: Proceedings of the 18th International Conference on Static Analysis. SAS'11. Berlin, Heidelberg: Springer-Verlag; 2011. p. 95–111.

[9] Staats M, Pâsâreanu C. Parallel Symbolic Execution for Structural Test Generation. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. ISSTA '10. New York, NY, USA: Association for Computing Machinery; 2010. p. 183–194. Available from: https://doi.org/10.1145/1831708.1831732.

[10] Zucker P. Formal methods for the informal engineer. Draper Labs; 2021. Available from: https://www.youtube.com/watch?v=56IIrBZy9Rc.

[11] Davis M, Putnam H. A Computing Procedure for Quantification Theory. J ACM. 1960 jul;7(3):201–215. Available from: https://doi.org/10.1145/321033.321034.

[12] Davis M, Logemann G, Loveland D. A Machine Program for Theorem-Proving. Commun ACM. 1962 jul;5(7):394–397. Available from: https://doi.org/10.1145/368273.368557.

[13] Horenovsky M. Modern sat solvers: Fast, neat and underused (part 1 of N). The Coding Nest; 2019. Available from: https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/.

[14] Gritman A, Ha A, Quach T, Wenger D. Conflict Driven Clause Learning. University of Washington;. https://cse442-17f.github.io/Conflict-Driven-Clause-Learning/.

[15] Gupta A. Lecture 06-1 SAT solver optimizations: 2-watched literals; 2020. Available from: https://www.youtube.com/watch?v=n3e-f0vMHz8&list=PLbLuy9jaJwu07biHdKGHLmuCYClD-2iHj&index=6.

[16] Tichy R, Glase T. Clause Learning in SAT; 2006. Available from: https://www.cs.princeton.edu/courses/archive/fall13/cos402/readings/SAT_learning_clauses.pdf.

[17] Torlak E. A Modern SAT Solver; 2021. Available from: https://courses.cs.washington.edu/courses/cse507/17wi/lectures/L02.pdf.

[18] Marques-Silva J. CDCL SAT Solving: Past, Present & Future; 2016. Available from: http://www.fields.utoronto.ca/sites/default/files/talk-attachments/marques-silva-fields16-talk.pdf.

[19] De Moura L, Bjørner N. Satisfiability modulo Theories: Introduction and Applications. Commun ACM. 2011 sep;54(9):69–77. Available from: https://doi.org/10.1145/1995376.1995394.

[20] Bjørner N, de Moura L, Nachmanson L, Wintersteiger C. Programming Z3 - Stanford CS theory. Microsoft Research;. Available from: https://theory.stanford.edu/~nikolaj/programmingz3.html.

[21] Z3. GitHub; 2022. https://github.com/Z3Prover/z3.

[22] Leino R. Dafny. GitHub; 2021. https://github.com/dafny-lang/dafny.

[23] Moskal M, Lal A, Lahiri S. Boogie: An intermediate verification language; 2019. Available from: https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/.

[24] Boogie Repo. GitHub; 2021. https://github.com/boogie-org/boogie.

[25] Leino KRM, Monahan R. Dafny Meets the Verification Benchmarks Challenge. In: Leavens GT, O'Hearn P, Rajamani SK, editors. Verified Software: Theories, Tools, Experiments. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 112-26.

[26] Leino KRM. Types in Dafny; 2015. Available from: http://leino.science/papers/krml243.html.

[27] KOENIG J, LEINO KRM. Getting started with Dafny: A guide - microsoft.com. Microsoft Research; 2016. Available from: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml220.pdf.

[28] Dafny Documentation: Lemmas and Induction. Dafny;. https://dafny.org/dafny/OnlineTutorial/Lemmas.html.

[29] Leino K. Basics of specification and verification. KRM Leino; 2018. Available from: https://www.youtube.com/watch?v=oLS_y842fMc.

[30] Herbert L, Leino KRM, Quaresma J. In: Meyer B, Nordio M, editors. Using Dafny, an Automatic Program Verifier. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 156-81. Available from: https://doi.org/10.1007/978-3-642-35746-6_6.

[31] Leino K, Ford R. Dafny reference manual.; 2022. Available from: https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef.html#2-lexical-and-low-level-grammar.

[32] Dickey T. Berkeley Yacc; 2022. Available from: https://invisible-island.net/byacc/.

[33] Team J. JFlex; 2020. Available from: https://www.jflex.de/.

[34] Husdon S, Petter M. CUP, LALR Parser Generator for Java; 2014. Available from: http://www2.cs.tum.edu/projects/cup/.

[35] Parr T. ANTLR. GitHub; 2021. https://github.com/antlr/antlr4.

[36] Parr T. Language Implementation Patterns. The Pragmatic Bookshelf; 2011.

[37] Parr T, Harwell S, Fisher K. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. 2014. Available from: https://www.antlr.org/papers/allstar-techreport.pdf.

[38] Tomassetti G. Improving the performance of an ANTLR parser; 2019. Available from: https://tomassetti.me/improving-the-performance-of-an-antlr-parser/.

[39] Parr T, Wüstholz. Fine-grained Caching of Verification Results. 2016. Available from: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml245.pdf.

[40] Meierhans J. Tesco shopper's plea to bring back till staff. 2022 May. Available from: https://www.bbc.co.uk/news/business-61467165.

[41] Fredrikson DM. Incremental Proof Development in Dafny; 2016. Available from: https://courses.cs.washington.edu/courses/cse507/17wi/lectures/L02.pdf.

[42] Selection Sort in Dafny; 2014. Available from: https://stackoverflow.com/questions/24591668/selection-sort-in-dafny.

[43] Leino R. Dafny, Lemmas and Induction. GitHub; 2021. https://dafny.org/dafny/OnlineTutorial/Lemmas.html.

[44] Dafny: rotated region of an array method verification; 2016. Available from: https://stackoverflow.com/questions/34734758/dafny-rotated-region-of-an-array-method-verification.

[45] (Dafny) Adding elements of an array into another - loop invariant; 2017. Available from: https://stackoverflow.com/questions/44217256/dafny-adding-elements-of-an-array-into-another-loop-invariant.

[46] Dafny: What does no terms found to trigger on mean?; 2018. Available from: https://stackoverflow.com/questions/49398650/dafny-what-does-no-terms-found-to-trigger-on-mean.