Architecture of Esc4: Languages and Visitors

Perry R. James   Patrice Chalin

September 24, 2008

Note to the reader: Because Esc4 is under active development, this document describes not only what has already been implemented but also *what we hope to soon have implemented*. To easily distinguish these, different fonts are used.

1 Introduction

Esc4 is to be Jml4's initial offering for static verification. The first stages of Esc4 can be thought of as a compiler from JML-annotated Java to Verification Conditions. Like many compilers, the architecture of Esc4 is a series of pipes and filters. The various stages transform the source, in our case the Jml-decorated JDT AST for a single method, first to a GC language then into VCs that can be processed by a theorem prover back-end, the `ProverCoordinator`. Our approach is based on that presented by Barnett and Leino in [**?**]. Finally, the results of the theorem provers is presented to the user in the form of errors and warnings. Each of the boxes in Fig. 1 is expanded in the following sections.

2 Guarded Command Translator

The purpose of the GC Translator is to convert the input AST to a desugared, passified, acyclic Guarded Command program. The first step is to remove the the first level of complexities of the JDT's AST by translating a method to a fully sugared Guarded Command (GC) program. Next, branching statements are removed to produce an acyclic GC program. A separate step removes any remaining sugar-

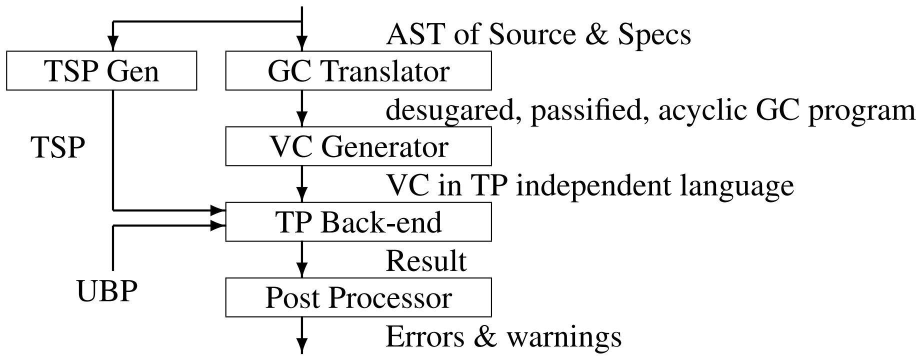

Figure 1: Top-level Architecture

ing before DSA is applied to produce a much simpler Control-Flow Graph (CFG) program `new name needed`.

2.1 From the JDT's AST to Our Own

The first step towards producing the final CFG program is to translate the JDT's AST to a highly sugared form that removes much of the information not needed for ESC. This is done by the `Ast2SugaredVisitor`. The main difference between this GC language and Dijkstra's [**?**] is that guards are replaced with `assume` statements. Except for a few constructs, this translation is almost a literal translation. Formal parameters are treated as any other local variable declaration, except their declarations come at the very beginning of the GC program and are followed by assuming the class's invariants and the method's preconditions.

Two blocks are produced in this process. The first is given the name `start` and consists of the declaration of the formal parameters, assuming the class's invariant and method's precondition, then the method body, and finally a `goto return`. The second is given the name `return` and consists of asserting the class's invariant and method's postcondition.

One kind of statement in the JDT's AST is a `Block`, which contains an array of statements. In the case this array is `null`, it is translated as `assert true` (also known as SKIP). When there is more than a single element in the array, each of the elements is converted, and the resulting list is folded into `SugaredSequence` statement.

In Java, some expressions (e.g., assignment expressions) can be a statements on their own. These are translated to `SugaredExprStatements`. Java supports several forms of assignments, including simple (`i = 3`), compound (`i += 3`), prefix (`++i`), postfix (e.g., `i++`). All but the last are translated into simple assignments as per the Java Language Specification [**?**], and postfix expressions are left in the various VC languages until passification (see below, Section 2.4).

`SingleNameReferences` and `CompoundNameReferences` are converted to representations of fields or local variables, as appropriate. Note that static fields are translated as variables, not fields.

All local variables and formal parameters are made unique by associating with them the location of their declaration.

`while` statements can be labeled statements in Java, so we provide a label for unlabeled loops. Similarly, labels are provided for unlabeled `break` and `continue` statements. Removing loops is made simpler if each `break` and `continue` all `SugaredBreak` and `SugaredContinue` statements have a label.

Conditional-And (`&&`) and Conditional-Or (`||`) Operator expressions are translated to Conditional Operator (`?:`) expressions.

The second step is to convert the fully sugared program to an acyclic Control Flow Graph with sugared GCs in the blocks that form the nodes.

2.2 Removing Control Flow Statements

Once we have a `start` block that contains the fully sugared version of the method body, we remove control-flow statements so that the resulting program is a set of blocks that end in `goto` statements. This transformation is based on the method presented by Barnett and Leino in "Weakest-Precondition of Unstructured Programs" and is done by the `DesugarLoopVisitor`. The main difference between the language used and Dijkstra's is that the choice operator is replaced with `goto`s and guards are replaced with `assume` statements.

Most of the statements and expressions are passed through unchanged. Sequences are handled in a special way because most of the constructs that are not passed through will be converted to blocks that end in `goto`s. If the first statement in a sequence is a `break` or a `continue`, then anything that follows is not meaningful, as it was added by this visitor while processing surrounding loop or `if` construct. New blocks are created to follow a loop or `if`.

A `SugaredIfStatement` is replaced with `goto [then, after]`, and new blocks are created for the two branches. The `then` block is prefaced with assuming the condition, and the `else` block is prefaced with assuming its negation. A missing else clause is replaced with `SKIP`. Both blocks end with `goto [afterIf]`, where `afterIf` contains the statements following the original `if` statement.

The following assumes the old loop semantics. A separate paper will describe the new semantics.
A `SugaredWhileStatement` is replaced with `assert invariant ; goto [header]`. New blocks are created for the loop header, the loop body, after the loop, and for a break target. The loop header havocs the targets of the loop (i.e., variables and fields that are assigned to in the loop body) and then goes to either the `body` or `after`. The `body` block assumes the loop condition and loop invariants, stores the value of any loop variant expressions, the loop body, asserts the loop invariant and checks that the variant functions decreased. It ends with a `goto []`, indicating that it is a dead end. In reality, it is only evaluated to ensure that the body restores the loop invariant and decreases any loop variants. For any loop variant that is given, we add an invariant that its value is nonnegative. The `after` block assumes the loop invariant and the negation of the loop condition and ends with a `goto [breakTarget]`. `break` statements are translated to `goto [breakTarget]`, and `continue` statements are translated to an assertion of the loop invariant, a check that the variant function decreases, and ends with a `goto []`.

Do-while, for, enhanced-for, and switch statements are yet to be imlemented, but their implementations should be similar to those of if and while.

`return` statements with a value are translated to the sequence
`result = expr ; goto [return]`, where `expr` is the value returned. Those without a value are simply translated as `goto [return]`.

The only variables that can be mentioned in the method's postcondition are the method's formal parameters. Each such variable `v` in is replaced with the expression
`old(v)`. `What about static fields?`

Method calls are not removed in this phase but during Passification (see Section 2.4).

2.3 Final Desugaring

Once a sugared, acyclic CFG has been produced, we are ready to desugar it before moving on to passification. *During desugaring, decisions are made of the kinds of warnings that should be reported or ignored.* See the paper on "Guarded Commands." `need a reference here to Escj16c.`

2.4 Passification

Passification using Dynamic Static Assignment is also discussed briefly by Barnett and Leino in [**?**], who give an optimization that in some cases reduces the number of incarnations needed for some variables. A first step in passification is performing a topological sort of the desugared program's blocks. An incarnation map is maintained that stores a mapping from assignables (i.e., variables, fields, and arrays) to the set of integers representing the incarnations with that assignable's latest value. When visiting each block, the incarnation maps from its immediate parents in the Control Flow Graph must be reconciled to ensure that the maximum incarnation for each assignable from all is the same. If it is not, then extra assignments must be added to make it so.

During conversion to a DSA form, all AST nodes of types that can be assigned to are given an exlicit incarnation, and assignments of the form

```
x := E
```

are replaced with assume statements of the form

```
assume x == E,
```

where x has an explicit incarnation. If the expression on the right-hand side itself has embedded assignments, these are passified and included as assumptions before the statement currently being visited. This is done by storing a list of assumptions (and assertions) that need to be made before the current statement.

As an example, consider the assignment expression with an incarnation map with `i` and `k` mapped to $\{0\}$.

```
i = (i++) * (--k) / m()
```

which we will evaluate to the following:

```
assume $i_1$ = $i_0$ + 1
assume $k_1$ = $k_0$ - 1
assert $m_{precondition}$
assume $m_{postcondition}$
assume $i_2$ = $i_0$ * $k_1$ / $m_{result}$
```

Method calls are similarly replaced with a new variable holding the result (or `true` if the method's return type is `void`. The declaration and initialization of new binding variables to hold the evaluation of the actual parameters are added to the list, followed by the assertion of the method's precondition (and any necessary invariants), with the method's formal parameters replaced with the new binding variables. This substitution is performed by a `SimpleSubstVisitor`. The assumption of the invariants and the postcondition are similarly added to the list. `Once we have proper Assignable Clauses, we should insert the effect of havocing a method's assignable between the assert and assume.` Notice that it is not necessary for there to be an explicit assignment to the result's variable, since it will be set in many ways in the assumption of the postcondition.

The only quantified expressions are currently the universal and existential. Declarations for their bound variables are added to the list. Before passifying the range and body of the quantified expression, the contents of the side-effects list are stored in a local variable, and the list is emptied. The range and body's "side effects," which can only be caused by method calls, are formed into an expression that is made to imply the body, giving a new body. The side-effects list is restored before returning the result of passifying the quantified expression, which is itself a quantified expression. In the case of the universal quantifier, the range is made to imply the new body, while for the existential, the range is conjoined to it.

Conditional expressions (i.e., those with the form `a ? b : c`) required a bit more work. After passifying each of its three parts, the side-effect list is stored to local variables. Copies of the incarnation map that results from passifying the condition are made so that identical maps can be used to passify the two alternative expressions. If either of these passification steps detects side effects, the incarnation maps must be reconciled as when joining to previous blocks. This reconciliation ensures that the maximum incarnation for each assignable on both branches is the same.

2.5 Final words on Generating the CFG Program

The steps in this subsection described the transformation from an AST to a desugared, passified, acyclic Guarded Command program. The only statements that remain in the language are for variable declarations, assertions, assumptions, sequencing, and gotos. The Following will discuss the tranformation to a Verification Condition.

3 VC Generation

The Verification Condition Generator converts a GC program to VCs. In comparison to the AST language, the GC language processed in this stage is minuscule. Using a weakest-precondition calculus calculus, we compute a verification condition for each method. A later stage will pass these to one or more theorem provers.

4 Theorem Prover Back-end

The theorem prover backed is an area that is open to many possible enhancements. We currently support Simplify, CVC3 and Isabelle. We also support multithreaded-distributed verification. See [**?**] and [**?**] for more details.

5 Post Processing

Once the VCs for a method have been either found to be valid or not, this information needs to be presented to the user. Jml4 already made use of Eclipse's `ProblemReporter` infrastructure to provide easy-to-access information of this type to the programmer. *We may want to explore the possibility of underlining Esc4's errors or warnings in green, to differentiate them from the normal red and yellow.*

6 Conclusions

This document was intended to give a brief overview of processing that goes on in each of the stages of Esc4. Esc4 is a quickly evolving system. There are many areas which need to be fleshed out, but we feel the basic framework is in place.