Using ESC/Java2

Architecture, Hints, and Tricks

Joseph Kiniry

Department of
Computer Science
University College Dublin

Using ESC/Java2 Effectively

- basic familiarity with ESC/Java2 is easy
  - it is automatic and behaves like a compiler
- but any non-trivial use quickly becomes *very* difficult and time-consuming
  - complexity of Java and JML semantics
  - limitations of logic
  - designed limitations of tool
  - limitations of Simplify theorem prover

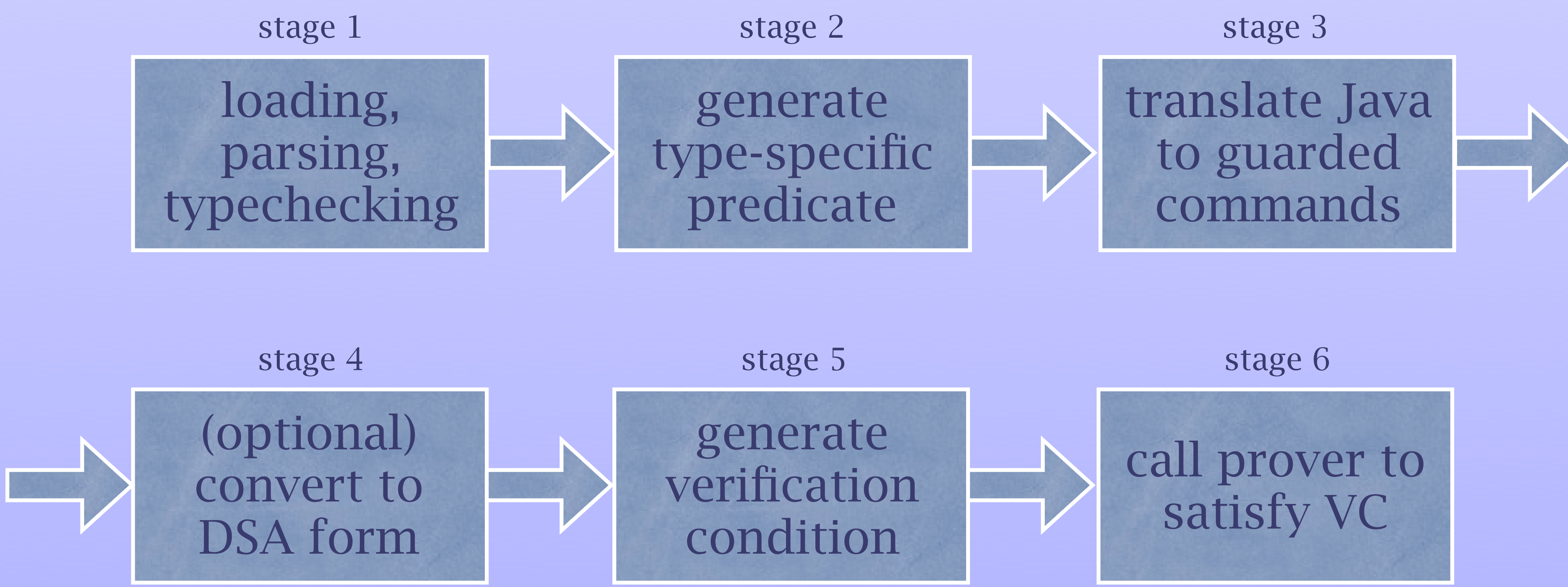
Thinking, not Hacking

- successful application of tool requires hard *thought* and very little *labor*
- recognizing that *specific misbehavior* implies *particular errors* in specifications or program code is key to effective use
- understanding *theoretical underpinnings* of extended static checking is very helpful
- a problem solving process for verification is needed for successful adoption

Verification Process

- the *key aspects* of the verification process
  - *small steps* in specification refinement and program development
  - *iterative* and *continuous* application of ESC/Java2 to method or class of focus
  - use a standardized problem-solving technique for resolving warnings
  - *think* before you *type*

ESC/Java2 Architecture



Department of
Computer Science
University College Dublin

How ESC/Java2 Works

- find, load, parse, and typecheck all relevant files and dependencies
  - this includes all refinements, models, etc.
- for each class being checked, generate a type-specific background predicate
  - type and subtype information about classes and fields
  - non_null-ness of references
  - size of constants

Department of
Computer Science
University College Dublin

How ESC/Java2 Works (2)

- translate each routine to be check into a verification condition (VC)
  - intermediate step in this translation is to translate Java into a (Dijkstra-like) guarded-command language
  - translation is accomplished by generating strongest-postconditions or weakest-preconditions for method body

How ESC/Java2 Works (3)

- ask theorem prover to prove VC
  - background predicate for Java expressed as a set of axioms
  - type-specific background predicate generated in second step is assumed true
  - assert VC is true
- if proof fails and prover finds counterexample, translate result back to warning message and Java, if possible

Examining the Results of Each Stage

- -v alone to print information on loading, parsing, refinement, etc.
- -showDesugaredSpecs to see heavyweight specifications desugared to lightweight ones (will also be -sds in next release)
- -pgc to print guarded command
- -ppvc to pretty-print verification condition
- -pxLog to print predicate sent to prover

A Stage-Driven Process

- ensure that the proper source and bytecode files are being loaded
  - this is particularly important when initially setting up a verification problem and when using refinement
- make sure that your specs mean what you think they do be examining the desugared specs
  - multiple heavyweight specs sometimes have unintuitive meaning for the beginner

A Stage-Driven Process (2)

- check size of “local contributors”
  - e.g., 35 types 99 invariants 62 fields
- examine the generated VC
  - it must has a reasonable structure
    - type-specific background predicate, followed by translated specification and program code
  - it is reasonably sized
    - ~1MB is ok, multiple MB is a problem

Datagroups and Ghost Fields

- datagroups are used to specify sets of fields that are interrelated
  - the primary datagroup used by non-expert ESC/Java2 users is *Object.objectState*
- ghost fields are specification-only fields that can be assigned using the *set* keyword
  - the primary ghost field used by non-experts is *Object.owner*

The Datagroup *Object.objectState*

```
public class Object {

    /** A data group for the state of this object.  This is used to
     * allow side effects on unknown variables in methods such as
     * equals, clone, and toString. It also provides a convenient way
     * to talk about "the state" of an object in assignable
     * clauses.
     */
    //@ public non_null model JMLDataGroup objectState;
    //@ represents objectState <- JMLDataGroup.IT;
```

The Ghost Field *Object.owner*

```
/** The Object that has a field pointing to this Object.
  * Used to specify (among other things) injectivity (see
  * the ESC/Java User's Manual).
  */
 /*@ ghost public Object owner;
                         in objectState;
   @*/
```

Dealing with Complexity

- specification and code complexity are the primary factors in verification complexity
- if performing “Design by Contract” then one can “Design for Verification” also
- if performing “Contract the Design” then verification is sometimes only possible with refinement if code modification is not permitted

Managing Spec Complexity

- write and verify specs *iteratively* using *very small steps*
- use *independent heavyweight specification blocks* to specify *independent behaviors*
- ensure your specs are *sound*
  - assert a false predicate to check
  - eliminate suspect predicates iteratively to determine source of unsoundness

Managing Spec Complexity (2)

- use ghost variables or model fields to factor out complex specification subexpressions
  - helps with comprehension, not verification
- avoid universally quantified expressions
- use the *objectState* datagroup as much as reasonable for your frame axioms
- use the *owner* field to disambiguate objects

Managing Code Complexity

- track cyclic complexity of method bodies
  - each branch, switch case, loop, and exception block doubles complexity
- decompose methods into smallest reasonable units
  - Smalltalk and Eiffel method size rule-of-thumb applies (e.g., all methods <15 LOC)
- avoid constructors that make calls

Department of
Computer Science
University College Dublin

Managing Code Complexity (2)

- focus on methods that make no calls first
- work from low to high cyclic complexity
- use assertions to check
- recognize sources of incompleteness of Java semantics
  - complex arithmetic
  - bit-level operations
  - String manipulations

Refinement for Complex Verification

- if you have a method with high cyclic complexity that you cannot refactor
  - inherit and override
  - implement and verify separate private methods for each branch of original method
  - implement overridden version as composition of verified new methods