ESCJ 16c:
Java to Guarded Commands translation

Edited by Rustan and Jim, most recently on 26 August 1998.
First draft by Rustan and Raymie, 11 April 1997.
Some additional comments from Cormac in italics.
Compaq Confidential.

ESCJ 16c:
Java to Guarded Commands translation....1
0Introduction....2
1Java-like AST....3
2Guarded command AST....5
  2.0Semantics of guarded commands 6
  2.1Semantics of loop....7
  2.2Semantics of call....10
3Special variables and literals....11
4Translating expressions....12
  4.0Assignment expressions.... 16
  4.1Method call expressions....19
5Translating specification expressions....20
6Translating statements....22
  6.0Translating loops....25
7Synthesizing method specifications....27
  7.0GetCombinedMethodDecl....29
    7.0.0Signature....29
    7.0.1Combining requires clauses....30
    7.0.2Combining modifies lists....31
    7.0.3Combining ensures clauses....31
  7.1FilterMethodDecl....31
  7.2 TrMethodDecl....32
    7.2.0Preconditions....32
    7.2.1Targets....33
    7.2.2Postconditions....33
  7.3ExtendSpecForCall....34
    7.3.0Adding preconditions.... 34
    7.3.1Adding postconditions....35
  7.4ExtendSpecForBody....35
    7.4.0Adding preconditions.... 35
    7.4.1Adding postconditions....36
8Verification conditions....36
  8.0Scope-specific background predicate....36
  8.1Methods and constructors....37
  8.2Static bodies....40

To do:

- Write static body VC gen.
- Define meta functions in a sensible order.
- Loop invariants (and/or “-Fast” translation of loops).
- The logic should assume labels to be distinct.
- Rename the guarded command **block** L: S **end** to something that doesn’t say “block” to avoid confusion with the Java **block** statement.
- Reconcile our AST with Raymie’s and Cormac’s.
- Pass over document for consistency (for example, make *LS* not be a keyword).

- Choose “freshly generated variable names” to produce meaningful counterexample contexts.
- Note that ESC/Java currently does not provide a way to monitor the contents of an array.
- Write down what we assume about *null* about fields of *null*. For example, *allocTime*(*null*) < *pre$alloc* and f[*null*] != *null* for **non_null** fields f. (Hm, note that our assumptions are not consistent in the initial state if two inconsistent invariants are declared (like x == 0 and x == 2), or if an invariant f == *null* is declared for a **non_null** field f.)
- State, in the ESC/Java Annotation Reference Manual, that the formal parameters of a method override are not allowed to mention the **non_null** modifier (instead, it implicitly inherits any **non_null** modifier of the corresponding formal parameter of the method being overridden). Also, state that the free variables of a **requires** clause of a method must be as visible as the method itself. Also say that an **ensures** clause of a constructor is allowed to mention *RES*. Also, state that a private field is allowed to be mentioned in a postcondition only if the method is final or private or if the enclosing class is final—otherwise, a syntactic warning is produced.
- Give location information for each **check** command.
- Introduce Java+ grammar for routine declarations, and use where appropriate.
- Reconcile error names with ESCJ 17.
- Update ESCJ 17 to use *max* instead of *min* for lock sets.
- Make index of meta functions.

0 Introduction

This note describes a translation of annotated Java into a guarded command-like language for the purpose of generating verification conditions. This note is not about resolution of names in Java, so we assume where convenient that names have been unique-ified. In particular, we assume the names of types, fields, and methods have been unique-ified. In the case of methods, unique-ification takes care of overloading by taking into account the types of formals.

We assume that we are given an AST (as described in section 1) for a Java method (or other body of code) to be checked, where all names (local variables, parameters, fields, methods, types) have already been resolved. Our goal is to produce a guarded command gc (as described in section 2) such that, with the background predicate (call it BG) produced according to ESCJ 8, *The logic of ESC/Java*, the condition

BG ==> *wlp*.gc.(*true*, *true*, *false*)

is valid if and only if m meets its specification. That is, the condition

BG && ! *wlp*.gc.(*true*, *true*, *false*)

is satisfiable if and only if m does not meet its specification. It behooves us now to explain *wlp* and "meets its specification".

For any guarded command gc and predicates N, X, and W, the predicate *wlp*.gc.(N, X, W) holds in exactly those initial states from which execution of gc either terminates normally in (a state satisfying) N, terminates exceptionally in X, goes wrong in W, or doesn't terminate at all. The computation of (an approximation to) *wlp* is described in section 2.

When we say that a Java method meets its specification, we have a particular notion in mind. This notion turns out to be unsound, because there are several kinds of errors that we don't check for. For one thing, we do not consider integer overflows or infinite loops to be violations of a method's specification. Also, we know that our treatment of **modifies** clauses is unsound, a choice we made in hope of making the ESC/Java tool easier to use without significantly undermining its ability to find errors. Furthermore, user-supplied annotations can introduce unsoundness, in both obvious ways--for example, by suppressing certain checks or introducing bogus assumptions--and in non-obvious ways--for example, by giving a lock order that is not a partial order.

Are there other sources of unsoundness? It would useful to have a document describing all the sources of unsoundness and incompleteness in the ESC/Java checker, including those introduced by the Java-to-GC translator, the VC generator, and the prover.

1 Java-like AST

We use italics for non-terminals and bold face for keywords. Sometimes we prefix a non-terminal with a descriptive comment (word) ending in an underscore. An asterisk denotes any number of occurrences of the immediately preceding terminal, non-terminal, or parenthesized construction.

We take the following non-terminals as primitives: *Identifier*, *Literal*, *UnaryOp*, *BinOp*.

Stmt ::=
  block *Stmt** **end**
| **var** *Modifier** *Identifier* [= *Expr*]
  Note that the *Expr* might be an array initializer expression. The *Identifier* introduced goes out of scope at the end of the innermost enclosing **block**, **for**, or **switch** statement, or method body.
| **label** *Identifier Stmt*
  The label is implicit in the address of the AST node. (Cormac's annotations are in this font).
| **skip**
| **eval** *Expr*
  Note that assignments are expressions, so the front end translates assignment statements into **eval** statements. Similarly, all method invocation statements are translated into **eval** statements. Note that the type of *Expr* is **void** if *Expr* is an invocation of a **void** method.
| **if** (*Expr*) *Stmt* **else** *Stmt*
  We assume that omitted **else** clauses have been replaced by **else skip**.
| *Identifier*: **while** (*Expr*) { **loop_invariant** *SpecExpr** } *Stmt*
  We assume that every **while**, **do**, and **for** statement has an explicit label, possibly provided by the front end.
  This label is implicit in the address of the AST node. After typechecking, the statement that is being aborted or continued **BreakStmt** *and* **ContinueStmt** *AST node can be retrieved via the static method* **FlowInsensitiveChecks.getBranchLabel**.
  The **LabelStmt** *AST node is therefore not used by the Java-to-GC translation.*
| *Identifier*: **do** { **loop_invariant** *Expr** } *Stmt* **while** (*Expr*)
| *Identifier*: **for** (*Stmt** ; *Expr* ; *Expr**) { **loop_invariant** *Expr** } *Stmt*
  The *Stmt** is the **for** initializer. It consists of either one **var** statement (whose scope is the entire **for** statement) or a list of expressions. The first *Expr* is the loop guard. The first *Expr** is a list of **for** update expressions. The final *Stmt* is the loop body.
| **break** *Identifier* | **continue** *Identifier*
  We assume that every **break** and **continue** statement has an explicit label, possibly provided by the front end.
  This label is implicit in the address of the AST node.
| **return** [*Expr*]
| **throw** *Expr*
| **try** *Stmt* **catch** (*Type Identifier Stmt*)* **end**
  Each *Type* specifies a class of exceptions for which the corresponding *Stmt* is a handler; the *Identifier* may be used in the corresponding *Stmt* to denote the exception caught.
| **try** *Stmt* **finally** *Stmt*
  We assume the front end translates the Java **try catch finally** statement into a **try finally** statement whose first component is a **try catch**.
| *Identifier*: **switch** (*Expr*) (**case** [*Expr*] *Stmt**)* **end**
  An omitted *Expr* means the **default** case. We assume there is exactly one **default** case. If the programmer doesn't supply one, the translation can add **case break** *Identifier* (where the label *Identifier* is the same as that of the **switch** statement) as the first or last case.
  The translation does not add the **default** *case, if one is not given in the source program.*
| **synchronized** (*Expr*) *Stmt*
  The *Expr* denotes an object treated as a mutex.
| **construct** *Identifier* (*Expr**)
  This statement is a constructor invocation (see *ExplicitConstructorInvocation* [*JLS*, 19.8.5]). It is called **ConstructorInvocation** in the ESC/Java front end.
  We assume that when the Java source for a constructor body for a proper subtype of *Object* does not begin with an explicit constructor invocation, the ESC/Java front end supplies the implicit constructor invocation as defined by [*JLS*, 8.6.5].
  The *Identifier* (which the type checker will already have disambiguated using the static types of **this** and of the *Expr**) names the superclass or sibling constructor to be called. The *Expr** are the arguments to that constructor.
  The Identifier is given via the **decl** *field of the* **MethodInvocation** *AST node.*
| **assert** *SpecExpr* | **assume** *SpecExpr*
  These statements come from ESC/Java annotations rather than from Java proper.
| **unreachable**

Expr ::=
  this
| *Literal*

| *Designator*
| *UnaryOp Expr*
| *Expr BinOp Expr*
| *Expr CondBinOp Expr*

A *CondBinOp* is one of conditional binary operators || and &&.

| (*Expr* ? *Expr* : *Expr*)
| **newarray** *Type Expr**

The *Type* specifies the element type of the array to be allocated. The *Expr** specifies the dimensions of the array to be allocated.

| **array** *Type Expr**

The *Type* specifies the element type of the array to be allocated and initialized. The *Expr** specifies the initial values of the elements, and the number of *Expr** indicates the length of the array to be allocated.

| *Expr* **instanceof** *Type*

The *Type* must be an object type.

| (*Type*) *Expr*

The *Type* is the type to which the *Expr* is to be cast.

| *Designator* = *Expr*

The language requires that the static type of the *Expr* be assignment covertible to the static type of the *Designator*. When assignment conversion may change the value of the right-hand side, for example when widening a `long` to a `float`, we assume that the ESC/Java front end supplies an explicit cast.

The front end does not supply the explicit case. We may want to add a brief intermediate pass that would do this.

| *Designator BinOp* = *Expr*
| *Designator BinOp*

The Java expression D++ falls into the *Designator BinOp* category, where the *BinOp* is + (which + it is depends on the type of D). The Java expression ++D is preprocessed into D += 1, and hence falls into the *Designator BinOp* = *Expr* category.
What happened with these expressions in Cormac and Raymie's AST?

The pre- and post- increment and decrement operators are translated into **UnaryExpr** *nodes, where the tag is* **INC** *or* **DEC** *for the pre- operators, and* **POSTFIXINC** *or* **POSTFIXDEC** *for the post- operators.*

| *MethodInvocation*

Why does Cormac and Raymie's AST contain a **ParenExpr** class? And what about the **AmbiguousVariableAccess**? Can we assume that these have been translated away by the front end by the time we get control?

The **ParenExpr** *class preserves information about where parens occurred in the source program. This information is useful for pretty-printing. The* **AmbiguousVariableAccess** *is removed by the name resolution pass, as is the* **AmbiguousMethodInvocation** *AST node.*

Designator ::=
Identifier

The *Identifier* denotes a local variable, parameter, or global variable.
Cormac and Raymie's AST has a class called **LocalVariableAccess**. Does that include parameters and globals?
The **LocalVariableAccess** *AST includes parameters and globals.*

| *Expr* . *Identifier*

The *Identifier* denotes a field.

| *Expr*[*Expr*]

There are four kinds of method invocations. In each case, we assume that the *Identifier* has been fully disambiguated.

MethodInvocation ::=

In all of these cases, the Identifier is given via the **decl** *field of the* **MethodInvocation** *AST node.*

new *Type Identifier* (*Expr**)

The *Type* specifies the class of the object to be allocated. The *Identifier* (which the type checker will already have disambiguated using the *Type* and the types of the *Expr**) names the constructor. The *Expr** are the arguments to the constructor.

| *Identifier* (*Expr**)

The *Identifier* names a static method. (The AST actually represents this form of method invocation as *Type*.*Identifier*(*Expr**), but this document can ignore the *Type*, because we assume that the *Identifier* has been fully disambiguated.)

| *Expr* . *Identifier* (*Expr**)

In this case, the *Identifier* may be either an instance method or a static method. If the *Identifier* names an instance method, then the result of evaluating the first *Expr* will be supplied as the actual self parameter to the method call. If the *Identifier* names a static method, then the *Expr* will be evaluated for side effects and the result discarded. In either case, the static type of the *Expr* will already have been used by the type checker to disambiguate the *Identifier*.

| **super** . *Identifier* (*Expr**)

In this case, too, the *Identifier* may be either an instance method or a static method. If the *Identifier* names an instance method, then **this** will be supplied as the actual self parameter to the method call. If the *Identifier* names a static method, then the keyword **super** is ignored by the translation. In either case, the direct superclass of the static type of **this** will already have been used by the type checker to disambiguate the *Identifier*.

SpecExpr ::=
  Expr
    This *Expr* must be side-effect free.
    Actually, this is really supposed to be an *Expr* in which any subexpression may be a *SpecExpr*.
| (**forall** (*Type Identifier*)* *SpecExpr*)
| (**exists** (*Type Identifier*)* *SpecExpr*)
| (**lblpos** *Label SpecExpr*)
| (**lblneg** *Label SpecExpr*)
| *PRE*(*SpecExpr*)
| *fresh*(*SpecExpr*)

Type ::=
  boolean | **byte** | **char** | **double** | **float** | **int** | **long** | **short**
| *Identifier*
    The *Identifier* is a declared class or interface, possibly a pre-declared name like *Object*.
| *Type*[]

2 Guarded command AST

Our translation targets a guarded command language whose syntax is given below.

We take the following non-terminals as primitives: *Identifier*, *Literal*, *UnaryOp*, *BinOp*, *Function*. The first four of these are supersets of the corresponding non-terminals in the Java AST. The guarded command non-terminal *variable* includes every Java *variable*, *field*, and *label*, as well as some variables introduced by the translation. The non-terminal *Function* includes the functions described in the Logic of ESC/Java. boy, this paragraph needs fixin'.

command ::=
  lhs = *rhs*
| **skip**
| **raise**
| **assert** *rhs*
| **assume** *rhs*
| **var** *variable** **in** *command* **end**
| *command* ; *command*
| *command* ! *command*
| *command* [] *command*
| **loop** { **inv** *condition** } *command* **end**
| **call** *MethodName* (*rhs**)

lhs ::=
  variable
| *variable*[*rhs*]
| *variable*[*rhs*][*rhs*]

rhs ::=
  lhs
| *Literal*
| *UnaryOp rhs*
| *rhs BinOp rhs*
| *Function* (*arg_rhs**)
| (**forall** $variable^{+}$ :: *rhs*)
| (**exists** $variable^{+}$ :: *rhs*)
| (**lblneg** *Identifier rhs*)
| (**lblpos** *Identifier rhs*)

condition ::=
  errorName, *location* : *rhs*

In the last line, *errorName* is the name *Free* or an error name as described ESCJ 17, *ESC/Java Annotation Reference Manual*, and *location* is Java source code location. If *errorName* is *Free*, the *location* field is not used and can be set to the null location.

> In many cases in this document, we have omitted the location of a condition triple. In those cases, the implicit location refers to a location near where the Java+ expression that is translated into the *rhs* is found. This document should probably make a precise choice of location explicit.

We define three deconstructor functions on conditions:

ErrorName[[EN, L : e]] == EN
Location[[EN, L : e]] == L
Predicate[[EN, L : e]] == e

In addition, we define the following shorthands:

if e **then** S0 **else** S1 **end** == (**assume** e ; S0 [] **assume** ! e ; S1)
block L: S **end** == (S ! **if ec** == L **then skip else raise end**)
raise L == (*EC* = L ; **raise**)
fail == **assume** *false*
modify lhs == **var** *x* **in** lhs = *x* **end**

where e is a *rhs*, S, S0, S1 are *command*, L is a *label*, and **ec** is a special variable introduced by the translation.

We also define a shorthand **check** whose grammar is:

check *location*, *condition*

and whose definition is:

check LUse, EN, LDecl : e ==
  #**if** (EN is *Free*)
    skip
  #**elsif** (checking of EN is enabled at LUse and at LDecl)
    assert (**lblneg** *MakeLabel*[[EN, LDecl, LUse]] e)
  #**else**
    assume e
  #**end**

where *MakeLabel* somehow concatenates its arguments into an identifier.

> In many cases in this document, we have omitted LUse. In those cases, the implicit location refers to a location near where the Java+ expression or statement that is translated into the **check** is found. This document should probably make a precise choice of location explicit.

2.0 Semantics of guarded commands

The commands are defined in terms of predicate transformers. For any command S and predicates N and X on the post-state of S, we define *ejp*[[S, N, X]] as a weak precondition sufficient to guarantee that any normally terminating execution of S establishes N, that any exceptionally terminating execution of S establishes X, and that no execution of S goes wrong. In particular, we have

ejp[[S, N, X]] ==> *wlp*.S.(N, X, *false*)

For any N, X, and W, we have:

wlp.(v = e).(N, X, W) == N[v   e]
wlp.(v[e0] = e).(N, X, W) == N[v   *store*(v, e0, e)]
wlp.(v[e0][e1] = e).(N, X, W) == N[v   *store*(v, e0, *store*(*select*(v, e0), e1, e)]

wlp.**skip**.(N, X, W) == N
wlp.**raise**.(N, X, W) == X
wlp.(**assert** e).(N, X, W) == (e && N) || (!e && W)
wlp.(**assume** e).(N, X, W) == (e ==> N)
wlp.(**var** v1 … vn **in** S **end**).(N, X, W) == (ALL v1 … vn :: *wlp*.S).(N, X, W))
(* provided v1 … vn are free in N, X, and W *)
wlp.(S0 ; S1,).(N, X, W) == *wlp*.S0.(*wlp*.(S1).(N, X, W), X, W)
wlp.(S0 ! S1,).(N, X, W) == *wlp*.S0.(N, *wlp*.S1.(N, X, W), W)
wlp.(S0 [] S1).(N, X, W) == *wlp*.S0.(N, X, W) && *wlp*.S1.(N, X, W)

For any N and X, we define:

ejp[[v = e, N, X]] == N[v e]
ejp[[v[e0] = e, N, X]] == N[v *store*(v, e0, e)]
ejp[[v[e0][e1] = e, N, X]] == N[v *store*(v, e0, *store*(*select*(v, e0), e1, e)]
ejp[[**skip**, N, X]] == N
ejp[[**raise**, N, X]] == X
ejp[[**assert** e, N, X]] == e && N
ejp[[**assume** e, N, X]] == (e ==> N)
ejp[[**var** v1 … vn **in** S **end**, N, X]] == (ALL v1 … vn :: *ejp*[[S, N, X]])
(* provided v1 … vn are free in N and X *)
ejp[[S0 ; S1, N, X]] == *ejp*[[S0, *ejp*[[S1, N, X]], X]]
ejp[[S0 ! S1, N, X]] == *ejp*[[S0, N, *ejp*[[S1, N, X]]]]
ejp[[S0 [] S1, N, X]] == *ejp*[[S0, N, X]] && *ejp*[[S1, N, X]]

Thus, any command composed only of assignment, **skip**, **raise**, **assert**, **assume**, **var**, ;, !, and [], we have:

ejp[[S, N, X]] == *wlp*.S.(N, X, *false*)

The semantics of the commands **loop** and **call** are more elaborate and are described next.

2.1 Semantics of loop

The predicate *wlp*.(**loop** { **inv** J1 … Jn } S **end**).(N, X, W) is defined as the weakest predicate P that satisfies the equation:

P == *wlp*.(**check** J1 ; … ; **check** Jn ; S).(P, X, W)

Since we don't have a way to compute arbitrary weakest fixpoints, we define the *ejp* of a loop by desugaring the loop into more primitive guarded commands.

ejp[[**loop** { **inv** J1 … Jn } S **end**, N, X]] ==
ejp[[*DesugarLoop*[[**loop** { **inv** J1 … Jn } S **end**]], N, X]]

ESC/Java features two ways to desugar **loop**, selected by a command-line switch:

DesugarLoop[[Loop, N, X]] ==
#**if** (`-loopsafe` is used)
DesugarLoopSafe[[Loop, N, X]]
#**else**
DesugarLoopFast[[Loop, N, X]]
#**end**

These satisfy, for any loop Loop, and any N, X, and W,

wlp.DesugarLoopSafe[[Loop]].(N, X, W) ==> *wlp*.Loop.(N, X, W)
wlp.Loop.(N, X, W) ==> *wlp.DesugarLoopFast*[[Loop]].(N, X, W)

We now define the two loop desugarings. The first is defined as follows.

DesugarLoopFast[[**loop** { **inv** J1 … Jn } S **end**]] ==
 CheckLoopInvariants[[J1 … Jn, "*Initially*", Loc]] ;
 S ;
 CheckLoopInvariants[[J1 … Jn, "*AfterIteration*", Loc]] ;
 fail

where Loc is the source location of the Java loop that gave rise to this **loop** command, and *CheckLoopInvariants* is defined as follows:

CheckLoopInvariants[[J1 … Jn, suffix, Loc]] ==
 check Loc, *ErrorName*[[J1]]suffix, *Location*[[J1]] : *Predicate*[[J1]] ;
 … ;
 check Loc, *ErrorName*[[Jn]]suffix, *Location*[[Jn]] : *Predicate*[[Jn]]

The other loop desugaring is defined as follows.

DesugarLoopSafe[[**loop** { **inv** J1 … Jn } S **end**]] ==
 (*CheckLoopInvariants*[[J1 … Jn, "*Initially*", Loc]] ; **fail**)
 [] (*Modify*[[*NTargets*[[S, {}]]]] ;
 assume *Predicate*[[J1]] ; … ; **assume** *Predicate*[[Jn]] ;
 S ;
 CheckLoopInvariants[[J1 … Jn, "*AfterIteration*", Loc]] ;
 fail)

where Loc is the source location of the Java loop that gave rise to this **loop** command, and where

Modify[[{v1 … vn}]] ==
 modify v1 ; … ; **modify** vn

Modify[[**bottom**]] ==
 fail

where **bottom** is a special "set" that satisfies the following properties, for any set of variables V (possibly **bottom** or {}):

- **bottom** ∪ V == V
- V ∪ **bottom** == V
- **bottom** – V == **bottom**

(It may seem from these properties that **bottom** equals {}. However, **bottom** is different, because *ShakeUp*, *NTargets*, and *XTargets* treat **bottom** and {} differently. For example, *Modify*[[{}]] == **skip** whereas *Modify*[[**bottom**]] == **fail**.)

Functions *NTargets* and *XTargets* take two arguments, a guarded command and a set of variables (possibly **bottom**), and return a set of variables (possibly **bottom**). Informally, *NTargets*[[S, V]] is the set of variables that can be modified as a result of a normal-outcome execution of the command S ; *Modify*[[V]] (where failing is considered not a normal-outcome execution). Similarly, *XTargets*[[S, V]] is the set of variables that can be modified as a result of a exceptional-outcome execution of the command S ! (*Modify*[[V]] ; **raise**) (where failing is considered not an exceptional-outcome execution). Here are their definitions: For any command C,

NTargets[[C, **bottom**]] == **bottom**
XTargets[[C, **bottom**]] == **bottom**

For any set of variables V other than **bottom**,

NTargets[[v = e, V]] == V ∪ {v}
NTargets[[v[e0] = e1, V]] == V ∪ {v}
NTargets[[v[e0][e1] = e2, V]] == V ∪ {v}
XTargets[[lhs = e, V]] == **bottom**

NTargets[[**skip**, V]] == V
XTargets[[**skip**, V]] == **bottom**

NTargets[[**raise**, V]] == **bottom**
XTargets[[**raise**, V]] == V

NTargets[[**assert** e, V]] == V
XTargets[[**assert** e, V]] == **bottom**

NTargets[[**assume** e, V]] == V
XTargets[[**assume** e, V]] == **bottom**

We can do a more precise job for *NTargets*[[**assume** e, V]], by returning **bottom** if e is *false*. Part of this slack is picked up by including **fail** as an actual AST node, rather than as sugar, and generating **fail** in the translation where we otherwise would have hardcoded **assume** *false*.

NTargets[[**fail**, V]] == **bottom**
XTargets[[**fail**, V]] == **bottom**

NTargets[[**var** v1 … vn **in** C **end**, V]] == *NTargets*[[C, V]] – {v1, …, vn}
XTargets[[**var** v1 … vn **in** C **end**, V]] == *XTargets*[[C, V]] – {v1, …, vn}

We require that v1 … vn not be elements of V.

NTargets[[C0 ; C1, V]] == *NTargets*[[C0, *NTargets*[[C1, V]]]]
XTargets[[C0 ; C1, V]] == *XTargets*[[C0, V]] ∪ *NTargets*[[C0, *XTargets*[[C1, V]]]]

NTargets[[C0 ! C1, V]] == *NTargets*[[C0, V]] ∪ *XTargets*[[C0, *NTargets*[[C1, V]]]]
XTargets[[C0 ! C1, V]] == *XTargets*[[C0, *XTargets*[[C1, V]]]]

NTargets[[C0 [] C1, V]] == *NTargets*[[C0, V]] ∪ *NTargets*[[C1, V]]
XTargets[[C0 [] C1, V]] == *XTargets*[[C0, V]] ∪ *XTargets*[[C1, V]]

NTargets[[**loop** { **inv** J1 … Jn} C **end**, V]] == **bottom**
XTargets[[**loop** { **inv** J1 … Jn} C **end**, V]] == *NTargets*[[C, *XTargets*[[C, V]]]] ∪ *XTargets*[[C, V]]

NTargets[[**call** m(e1 … en), V]] == V ∪ *Domain*[[wt]]
XTargets[[**call** m(e1 … en), V]] == V ∪ *Domain*[[wt]]

where wt is the whole-targets map in the **whole-targets** clause of the method specification returned by *GetSpecForCall*[[m, Sc]], where Sc is the current scope (see below for all of these definitions).

Note. A simpler definition of *NTargets* that seems good enough for the first cut of ESC/Java is the following. Note that there is then no need for the *XTargets* function or the **bottom** value.

NTargets[[S, V]] == *SimpleTargets*[[S]] ∪ V

SimpleTargets[[v = e]] == {v}
SimpleTargets[[v[e0] = e1]] == {v}
SimpleTargets[[v[e0][e1] = e2]] == {v}
SimpleTargets[[**skip**]] == {}
SimpleTargets[[**raise**]] == {}
SimpleTargets[[**assume** e]] == {}
SimpleTargets[[**fail**]] == {}
SimpleTargets[[**var** v1 … vn **in** C **end**]] == *SimpleTargets*[[C]] – {v1, …, vn}

SimpleTargets[[C0 ; C1]] == *SimpleTargets*[[C0]] ∪ *SimpleTargets*[[C1]]
SimpleTargets[[C0 ! C1]] == *SimpleTargets*[[C0]] ∪ *SimpleTargets*[[C1]]
SimpleTargets[[C0 [] C1]] == *SimpleTargets*[[C0]] ∪ *SimpleTargets*[[C1]]
SimpleTargets[[C0 ; C1]] == *SimpleTargets*[[C0]] ∪ *SimpleTargets*[[C1]]
SimpleTargets[[**loop** { **inv** J1 … Jn} C **end**]] == *SimpleTargets*[[C]]
SimpleTargets[[**call** m(e1 … en)]] == *Domain*[[wt]]

where wt is as described above.

2.2 Semantics of call

The semantics of **call** m(e1 … en) depends on the *method specification* associated with m in the scope where the call appears. A *scope* is the set of declarations visible from a given class (or interface). To describe the semantics of **call**, we will in this section describe the abstract syntax of a method specification and the desugaring of a **call** command into more primitive commands. Later in this document (section 7), we describe how the method specification is computed in a given scope.

A method specification has the form:

> **spec** T m(p1 … pn) **throws** {X1 … Xx}
> **precondition** P1 … **precondition** Pj
> **targets** D1 … Dk
> **whole-targets** wt
> **postcondition** Q1 … **postcondition** Qm

In this method specification:

- m is a method name.
- p1 … pn are formal parameter names, possibly including the special name *this*.
- T is a result type or **void**.
- P1 … Pj are *conditions* whose predicates' free variables are drawn from the top-level program variables (fields, static fields, and special program variables like *elems* and *alloc*) and p1 … pn. Those conditions whose error name is *Free* are called *free preconditions*; the others are called *checked preconditions*.
- D1 … Dk are designator expressions (that is, guarded command *lhs's*) known as *target designators*.
- wt is a map from variables to variables. In particular, wt maps the set of variables one gets from "shaving" the target designators D1 … Dk to similar variables adorning with the suffix @pre (see section 7.2.0). "Whole targets" refers to the variables in the domain of wt.
- Q1 … Qm are conditions, the free variables of whose predicates are drawn from the top-level program variables, p1 … pn, the variables in image of wt, and the special result variables *EC*, *RES*, and *XRES*. Those conditions whose error name is *Free* are called *free postconditions*; the others are called *checked postconditions*.

We consider p1 … pn and the variables in the image of wt to be bound within the method specification. All other free variables are either static fields, instance variables, or special variables. The variables in the image of wt do not occur in the list of target designators, despite the fact that any index expression occurring in a target designator refers to the value of the index expression in the pre-state.

Usually, the formal parameter names p1 … pn correspond to formal parameters declared by the programmer. For instance methods, however, p1 is the special variables *this* and p2 … pn correspond to the formal parameters declared by the programmer.

All preconditions P1 … Pj are assumed on entry to implementations, but only the checked preconditions are checked at call sites. The free preconditions need not be checked at call sites because they are guaranteed by the Java type system and other checking mechanisms (like **non_null**). Similarly, all postconditions Q1 … Qm are assumed after calls, but only the checked postconditions are checked on exit from implementations. The free postconditions need not be checked on exit from implementations because they are guaranteed by the Java type system and other checking mechanisms.

Given that m is a method name associated in the current scope with the template method specification above, the command **call** m(e1 … en) occurring at a location L desugars as follows:

var p1@L … pn@L **in**
p1@L = e1 ; … ; pn@L = en ;

check L, P1 ; … ; **check** L, Pj ;
Note that the **check** desugars to **skip** for free preconditions.

var pt[[*Image*[[wt]]]] **in**
#**for** w **in** *Domain*[[wt]] **do**
assume pt[[wt[[w]]]] == w ;
Do we get better performance if instead of this assumption we do the semantically equivalent assignment pt[[wt[[w]]]] = w ?
#**end**

modify *IndexSubst*[[D1, pt]] ; … ; **modify** *IndexSubst*[[Dk, pt]] ;
modify *EC* ; **modify** *RES* ; **modify** *XRES* ;

assume pt[[*Predicate*[[Q1]]]] ; … ; **assume** pt[[*Predicate*[[Qm]]]] ;

#**if** ({X1 … Xx} is nonempty)
The reason for producing the following command only conditionally is one of concern for performance: It would be correct, but we conjecture inefficient, to always emit the following command.
(**assume** *EC* == *ec$return* [] **assume** EC == *ec$throw* ; **raise**)
#**end**
end
end

in which pt is the map {p1   p1@L, …, pn   pn@L} ∪ *Remap*[[wt, L]], where

Remap[[wt, L]] ==
#**for** w **in** *Domain*[[wt]] **do**
{wt[[w]]   w@L} ∪
#**end**

And where *IndexSubst* is defined as follows:

IndexSubst[[g, pt]] ==
g

IndexSubst[[f[E], pt]] ==
f[pt[[E]]]

IndexSubst[[e[E0][E1] , pt]] ==
e[pt[[E0]]][pt[[E1]]]

3 Special variables and literals

The translation introduces several special variables and literals.

The special variables *EC* (exception code), *RES*, and *XRES* are used is the translation of **return**, **throw**, **break**, **continue**, and method calls, all of which give rise to uses of the guarded command **raise**. By convention, the guarded commands generated by the translation always set *EC* (and possibly *RES* or *XRES*) before performing a **raise**. The enclosing exception handler (that is, the command T in S ! T) then tests *EC* (and possibly *RES* or *XRES*) when determining how to proceed.

More specifically, before a **raise** that corresponds to a Java **return**, the guarded command sets *EC* to the special literal *ec$return* and sets *RES* to the return value, if there is one. Before a **raise** that corresponds to a Java **throw**, the guarded command sets *EC* to the special literal *ec$throw* and sets *XRES* to the exception thrown. The translation of a method call uses a combination of these. Before a **raise** that corresponds to a Java **break** L, the guarded command sets *EC* to L. Finally, before a **raise** that corresponds to a Java **continue** L, the guarded command sets *EC* to continue$L, which is a name derived from the name L.

The special variable *elems* models the state of all arrays.

The special variable *alloc* represents the current allocation time.

The special variable *LS* represents the set of locks held by the current thread.

4 Translating expressions

In this section, we describe the translation of Java expressions. Since Java expressions may have side effects and guarded command expressions must not, it is occasionally necessary to introduce temporary variables. For example, the Java expression

```
x += (x = 3);
```

may be translated into the guarded command

var *oldx* **in** *oldx* = *x* ; *x* = 3 ; *x* = *oldx* + *x* **end**

Our translation introduces temporary variables where these may be useful. Throughout the translation, we assume that the temporary variables introduced have fresh names; the choice of these names may affect the readability of satisfying assignments, but we not address that issue here.

In this section, we define a translation function *TrExpr* for expressions. The signature of this function is *TrExpr*[[E, p, V, r]], where E is a Java expression, p is a set of *protect expressions* (defined below), V is a set of temporary variable names, and r is a guarded command expression. E and p are in-parameters, V is an in-out-parameter, and r is an out-parameter. *TrExpr*[[E, p, V, r]] returns a guarded command C that essentially evaluates the side effects of E, raises any exception raised by E, and causes any error of E. This command may include assignments to freshly generated temporary variables; as a side effect, *TrExpr* adds these temporary variables to V. Another side effect of *TrExpr* is to set r to an guarded command expression whose value in the normal post-state of C corresponds to the Java value of E. The expression r has the property of being insensitive to side effects of any protect expression in p.

A protect expression is either a Java expression or something of the form F=, where F is map variable. An expression e is insensitive to side effects of a Java expression E when no normally terminating evaluation of E can change the value of e. An expression e is insensitive to side effects of F= when it is insensitive to arbitrary modifications of F.

Before defining *TrExpr*, we describe three subroutines of which we will make frequent use, *Clash*, *Impure*, and *Protect*.

For any guarded command expression e and any protect expression q, *Clash*[[e, q]] must be *true* if e is sensitive to any side effect of q, but is allowed to be *true* more often than that. For now, *Clash* is conservatively defined as follows: For any Java expression E and map variable F,

Clash[[e, E]] ==
(e mentions any Java non-final local variable, non-final field, *elems*, *alloc*, or *RES*) && *Impure*[[E]]

Clash[[e, F=]] ==
(e mentions F)

In future versions of ESC/Java, we may use a more aggressive definition of *Clash*.

For any Java expression E, *Impure*[[E]] is *true* if E contains any
- assignment (=, +=, etc.),

- pre-increment, pre-decrement, post-increment, post-decrement (++ or --),
- object creation (**new**), or
- method invocation.

Note that the possibility of raising an exception or going wrong does not imply that an expression is impure; only state changes do.

The signature of *Protect* is *Protect*[[e, p, V, r]], where e is a guarded command expression, and p, V, and r are as in the signature of *TrExpr*. In a nut shell, *Protect* sets r to an expression that is equivalent to e, but is insensitive to side effects of the protect expressions in p. In doing so, it may make use of a temporary variable *v*, which it adds to V, and generate (i.e., return) a guarded command that assigns the value e to *v*.

Protect[[e, {E1 … En}, V, r]] ==
  #**if** (*Clash*[[e, E1]] || … || *Clash*[[e, En]])
    #V = V *v* ;
    #r = *v* ;
    v = e ;
  #**else**
    #r = e;
  #**end**

An explanation of our notation is in order. We use assignment statements where the left-hand side begins with a # to denote meta-assignments. Variables type set in italics denote fresh guarded command variables. Lines that don't begin with # (like the assignment *v* = e; in this example) show a guarded command fragment returned by *Protect*.

> The actual ESC/Java implementation simply uses booleans where we use sets of protect expressions. Where we have written a set {p1 … pn} as a protect argument to *TrExpr*, the implementation passes the boolean *Impure*[[p1]] || … || *Impure*[[pn]], where *Impure*[[pi]] is defined as described above if pi is a Java expression and as *true* if pi has the form F=. Since *TrExpr* usually passes its protect argument to *Protect*, the actual ESC/Java implementation uses a boolean for this parameter, too, and implements *Protect* as follows:
>
> *Protect*[[e, p, V, r]] ==
>   #**if** (p && (e mentions any Java non-final local variable, non-final field, *elems*, or *alloc*))
>   #V = V *v* ;
>   #r = *v* ;
>   *v* = e ;
>   #**else**
>   #r = e;
>   #**end**

In the translation below, we use x, xj to denote variables, E, Ej to denote Java expressions, C to denote any literal, and T to denote a type.

TrExpr[[**this**, p, V, r]] ==
  #r = *this*

TrExpr[[C, p, V, r]] ==
  #r = C

TrExpr[[x, p, V, r]] ==
  ReadCheck[[x]] ;
  Protect[[x, p, V, r]]

where

ReadCheck[[x]] ==
  #**if** (x declared with **uninitialized**)
    check *InitializationViolation* : init$x ;
  #**end**

#**if** (x declared with **defined_if** P)
check *DefinednessViolation* : *TrSpecExpr*[[P]] ;
#**end**
#**if** (x declared with the **monitored_by** expressions MU1 … MUn where 0 < n)
check *SharingViolation* :
(*TrSpecExpr*[[MU1]] != *null* && *LS*[*TrSpecExpr*[[MU1]]]) ||
… ||
(*TrSpecExpr*[[MUn]] != *null* && *LS*[*TrSpecExpr*[[MUn]]]) ;
#**end**

TrExpr[[E0[E1], p, V, r]] ==
#**var** e0 e1 **in**
TrExpr[[E0, {E1}, V, e0]] ;
TrExpr[[E1, {}, V, e1]] ;
ArrayAccessCheck[[e0, e1]] ;
Protect[[*elems*[e0][e1], p, V, r]]
#**end**

where

ArrayAccessCheck[[e0, e1]] ==
check *NullPointerException* : e0 != *null* ;
check *IndexOutOfBoundsExceptionLower* : 0 <= e1 ;
check *IndexOutOfBoundsExceptionUpper* : e1 < *array$length*(e0)

TrExpr[[E.F, p, V, r]] ==
#**var** e **in**
TrExpr[[E, {}, V, e]] ;
check *NullPointerException* : e != *null* ;
ReadCheck[[F[e]]] ;
Protect[[F[e], p, V, r]]
#**end**

where

ReadCheck[[F[e]] ==
#**if** (F declared with **defined_if** P)
check *DefinednessViolation* : *TrSpecExpr*[[P, {*this* e}, {}]] ;
#**end**
#**if** (F declared with the **monitored** and **monitored_by** expressions MU1 … MUn where 0 < n)
check *SharingViolation* :
(*TrSpecExpr*[[MU1, {*this* e}, {}]] != *null* && *LS*[*TrSpecExpr*[[MU1, {*this* e}, {}]]]) ||
… ||
(*TrSpecExpr*[[MUn, {*this* e}, {}]] != *null* && *LS*[*TrSpecExpr*[[MUn, {*this* e}, {}]]]) ;
#**end**

TrExpr[[unaryOp E, p, V, r]] ==
#**var** e **in**
TrExpr[[E, {}, V, e]] ;
Protect[[unaryOp(e), p, V, r]]
#**end**

TrExpr[[E0 binOp E1, p, V, r]] ==
#**var** e0 e1 **in**
TrExpr[[E0, {E1}, V, e0]] ;
TrExpr[[E1, {}, V, e1]] ;
#**if** (binOp is integer / or integer %)

```
    check ArithmeticException :  e1 != 0 ;
   #end
   Protect[[ binOp(e0, e1), p, V, r ]]
 #end
```

```
TrExpr[[ E0 || E1, p, V, r ]] ==
 #var e0 e1 in
   TrExpr[[ E0, {E1}, V, e0 ]] ;
   if ! e0 then
     TrExpr[[ E1, {}, V, e1 ]]
   end ;
   Protect[[ bool$or(e0, e1), p, V, r ]]
 #end
```

```
TrExpr[[ E0 && E1, p, V, r ]] ==
 #var e0 e1 in
   TrExpr[[ E0, {E1}, V, e0 ]] ;
   if e0 then
     TrExpr[[ E1, {}, V, e1 ]]
   end ;
   Protect[[ bool$and(e0, e1), p, V, r ]]
 #end
```

```
TrExpr[[ (E0 ? E1 : E2), p, V, r ]] ==
 #var e0 e1 e2 in
   TrExpr[[ E0, {E1, E2}, V, e0 ]] ;
   if e0 then
     TrExpr[[ E1, {}, V, e1 ]]
   else
     TrExpr[[ E2, {}, V, e2 ]]
   end ;
   Protect[[ term$conditional(e0, e1, e2), p, V, r ]]
 #end
```

```
TrExpr[[ newarray T E1 E2 … En, p, V, r ]] ==
 #var e1 e2 … en in
   TrExpr[[ E1, {E2 … En}, V, e1 ]] ;
   TrExpr[[ E2, {E3 … En}, V, e2 ]] ;
   … ;
   TrExpr[[ En, {}, V, en ]] ;
   #V = V a alloc′ ;
   assume array$fresh(a, alloc, alloc′, elems,
                              shapeMore(e1, shapeMore(e2, …(shapeOne(en))…)),
                              array(array(…(array(T))…)), zero) ;
   alloc = alloc′ ;
   #r = a
 #end
```

The number of applications of *array* around T in this assumption is n. The meta variable zero denotes the zero-equivalent value for type T.

```
TrExpr[[ array T E1 E2 … En, p, V, r ]] ==
 #var e1 e2 … en in
   TrExpr[[ E1, {E2 … En}, V, e1 ]] ;
   TrExpr[[ E2, {E3 … En}, V, e2 ]] ;
   … ;
   TrExpr[[ En, {}, V, en ]] ;
   #V = V a alloc′ ;
```

```
  assume alloc < vAllocTime(a) && vAllocTime(a) < alloc′ ;
  assume a != null && typeof(a) == array(T) && array$length(a) == n ;
  assume elems[a][0] == e1 && … && elems[a][n-1] == en ;
  alloc = alloc′ ;
  #r = a
 #end
```

```
TrExpr[[ E instanceof T, p, V, r ]] ==
 #var e in
  TrExpr[[ E, {}, V, e ]] ;
  Protect[[ is(e, T), p, V, r ]]
 #end
```

```
TrExpr[[ (T) E, p, V, r ]] ==
 #var e in
  TrExpr[[ E, {}, V, e ]] ;
  #if (T is an object type)
   check ClassCastException :  is(e, T)
   Protect[[ e, p, V, r ]]
  #else
   Protect[[ cast(e, T), p, V, r ]]
  #end
 #end
```

4.0 Assignment expressions

There are three kinds of assignment operators, namely direct assignment (as in *x* = 6), update assignment (as in *x* += 6), and post-update assignment (as in *x*++). There are also three kinds of l-values, namely variables (as in *x* = 6), instance variables (as in *o.f* = 6), and array elements (as in *a*[*i*] = 6). So, all in all, we consider nine cases. This results in some duplication, but we felt that this would increase clarity (and besides, 3*3 is not that much larger than 3+3).

```
TrExpr[[ x = E, p, V, r ]] ==
 #var e in
  TrExpr[[ E, {}, V, e ]] ;
  WriteCheck[[ x, e ]] ;
  x = e ;
  #if (x declared with uninitialized)
   init$x = bool$true ;
  #end
  Protect[[ x, p, V, r ]]
 #end
```

This comes from [*JLS*, 15.25.1].

where

```
WriteCheck[[ x, e ]] ==
 #if (x declared with non_null)
  check NullAssignmentViolation :  e != null ;
 #end
 #if (x declared with the monitored_by expressions MU1 … MUn where 0 < n)
  check SharingViolation :
     (TrSpecExpr[[ MU1 ]] != null || … || TrSpecExpr[[ MUn ]] != null) &&
     (TrSpecExpr[[ MU1 ]] == null || LS[TrSpecExpr[[ MU1 ]] ] ) &&
     … &&
     (TrSpecExpr[[ MUn ]] == null || LS[TrSpecExpr[[ MUn ]] ] ) ;
 #end
```

```
TrExpr[[ x binOp= E, p, V, r ]] ==
 #var old e in
   ReadCheck[[ x ]] ;
   Protect[[ x, {E}, V, old ]] ;
   TrExpr[[ E, {}, V, e ]] ;
   #if (binOp is integer / or integer %)
     check ArithmeticException :  e != 0 ;
   #end
   #if (range type of binOp is the static type of x)
     WriteCheck[[ x, binOp(old, e) ]] ;
     x = binOp(old, e) ;
   #else
     WriteCheck[[ x, cast(binOp(old, e), T) ]] ; // where T denotes the static type of x
     x = cast(binOp(old, e), T) ;
   #end
 #end
 Protect[[ x, p, V, r ]]
```

This comes from [*JLS*, 15.25.2].
Note that we need not set init$x to *true*, since the *ReadCheck* has checked that it is already *true*.
Section [*JLS*, 15.25] says that the result of this assignment expression is the value of the variable after the assignment has occurred; hence, we return x instead of binOp(*old*, e).

```
TrExpr[[ x binOp, p, V, r ]] ==
 ReadCheck[[ x ]] ;
 #V = V old ;
 old = x ;
 WriteCheck[[ x, binOp(x, 1) ]] ;
 x = binOp(x, 1) ;
 #r = old
```

Note that we need not set init$x to *true*, since the *ReadCheck* has checked that it is already *true*.
Note that this translation ignores the possibility of wrap-around.

```
TrExpr[[ E0.F = E1, p, V, r ]] ==
 #var e0 e1 in
   TrExpr[[ E0, {E1, F=}, V, e0 ]] ;
   check NullPointerException :  e0 != null ;
   TrExpr[[ E1, {}, V, e1 ]] ;
   WriteCheck[[ F[e0], e1 ]] ;
   F[e0] = e1;
   Protect[[ F[e0], p, V, r ]]
 #end
```

The ordering of the checks is spelled out in [*JLS*, 15.25.1].

where

```
WriteCheck[[ F[e0], e1 ]] ==
 #if (F declared with non_null)
   check NullAssignmentViolation :  e1 != null ;
 #end
 #if (F declared with the monitored and monitored_by expressions MU1 … MUn where 0 < n)
   check SharingViolation :
         (TrSpecExpr[[ MU1, {this    e0}, {} ]] != null || … || TrSpecExpr[[ MUn, {this    e0}, {} ]] != null)
         &&
         (TrSpecExpr[[ MU1, {this    e0}, {} ]] == null || LS[ TrSpecExpr[[ MU1, {this    e0}, {} ]] ]) &&
         … &&
         (TrSpecExpr[[ MUn, {this    e0}, {} ]] == null || LS[ TrSpecExpr[[ MUn, {this    e0}, {} ]] ]) ;
 #end
```

```
TrExpr[[ E0.F binOp = E1, p, V, r ]] ==
 #var e0 old e1 in
   TrExpr[[ E0, {E1, F=}, V, e0 ]] ;
   check NullPointerException :  e0 != null ;
   ReadCheck[[ F[e0] ]] ;
   Protect[[ F[e0], {E1}, V, old ]] ;
   TrExpr[[ E1, {}, V, e1 ]] ;
   #if (binOp is integer / or integer %)
     check ArithmeticException :  e1 != 0 ;
   #end
   #if (range type of binOp is the static type of E0.F)
     WriteCheck[[ F[e0], binOp(old, e1) ]] ;
     F[e0] = binOp(old, e1) ;
   #else
     WriteCheck[[ F[e0], cast(binOp(old, e1), T) ]] ; // where T denotes the static type of E0.F
     F[e0] = cast(binOp(old, e1), T) ;
   #end
   Protect[[ F[e0], p, V, r ]]
 #end
```

This comes from [*JLS*, 15.25.2].

```
TrExpr[[ E.F binOp, p, V, r ]] ==
 #var e in
   TrExpr[[ E, {F=}, V, e ]] ;
   check NullPointerException :  e != null ;
   #V = V old ;
   ReadCheck[[ F[e] ]] ;
   old = F[e] ;
   WriteCheck[[ F[e], binOp(old, 1) ]] ;
   F[e] = binOp(old, 1) ;
   #r = old
 #end
```

Note that this translation ignores the possibility of wrap-around.

```
TrExpr[[ E0[E1] = E2, p, V, r ]] ==
 #var e0 e1 e2 in
   TrExpr[[ E0, {E1, E2, elems=}, V, e0 ]] ;
   TrExpr[[ E1, {E2, elems=}, V, e1 ]] ;
   TrExpr[[ E2, {}, V, e2 ]] ;
   ArrayAccessCheck[[ e0, e1 ]] ;
   #if (static element type of E0 is a non-final object type)
     check ArrayStoreException :  is(e2, elemType(typeof(e0))) ;
   #end
   elems[e0][e1] = e2 ;
   Protect[[ elems[e0][e1], p, V, r ]]
 #end
```

The order of evaluation and checking (in particular, that E2 is evaluated before before any array access check is done) is specified in [*JLS*, 15.25.1]. Note that this is different from the order in which evaluation and checking is done for E0.F = E1, see above. It is also different from the order in which this check is done in the next case, E0[E1] binOp= E2 [*JLS*, 15.25.2]. The reason for this wisdom is unbeknownst to us.

```
TrExpr[[ E0[E1] binOp = E2, p, V, r ]] ==
 #var e0 e1 old e2 in
   TrExpr[[ E0, {E1, E2, elems=}, V, e0 ]] ;
   TrExpr[[ E1, {E2, elems=}, V, e1 ]] ;
   ArrayAccessCheck[[ e0, e1 ]] ;
   Protect[[ elems[e0][e1], {E2}, V, old ]] ;
   TrExpr[[ E2, {}, V, e2 ]] ;
   #if (binOp is integer / or integer %)
```

    check *ArithmeticException* : e2 != 0 ;
  #**end**
  #**if** (range type of binOp is the static type of E0[E1])
    elems[e0][e1] = binOp(old, e2) ;
  #**else**
    elems[e0][e1] = *cast*(binOp(old, e2), T) ; // where T denotes the static type of E0[E1]
  #**end**
  Protect[[*elems*[e0][e1], p, V, r]]
#**end**

This comes from [*JLS*, 15.25.2].

TrExpr[[E0[E1] binOp, p, V, r]] ==
 #**var** e0 e1 **in**
  TrExpr[[E0, {E1, *elems*=}, V, e0]] ;
  TrExpr[[E1, {*elems*=}, V, e1]] ;
  ArrayAccessCheck[[e0, e1]] ;
  #V = V *old* ;
  old = *elems*[e0][e1] ;
  elems[e0][e1] = binOp(*old*, 1) ;
  #r = *old*
 #**end**

Note that this translation ignores the possibility of wrap-around.

4.1 Method call expressions

Java features a number of different call expressions, namely instance method calls, static method calls, and constructor calls. (There are also constructor call statements. These will be described in the section 6 on statements.)

The abstract syntax of an instance method is:
    Expr . *Identifier* (*Expr**)
We treat the *Expr* before the "." as a parameter of the call.

The abstract syntax of a static method call can be one of:
    Identifier (*Expr**)
    Expr . *Identifier* (*Expr**)
    super . *Identifier* (*Expr**)
In the two latter cases, what goes before the "." is not a parameter of the call.

The abstract syntax of a class instance creation expression [*JLS*, 15.8] is:
    new *Type Identifier* (*Expr**)
We treat this simply as a constructor invocation.

The translation of a method invocation or constructor invocation emits a code fragment containing a **call** command.

TrExpr[[m (E1 E2 … En), p, V, r]] ==
 #**var** e1 … en **in**
  TrExpr[[E1, {E2 … En}, V, e1]] ; … ; *TrExpr*[[En, {}, V, en]] ;
  call m(e1 … en) ;
  Protect[[*RES*, p, V, r]]
 #**end**

TrExpr[[E0 . m (E1 E2 … En), p, V, r]] ==
 #**var** e0 … en **in**
  #**if** (m is a static method)
    TrExpr[[E0, {}, V, e0]] ;
  #**else**
    TrExpr[[E0, {E1 … En}, V, e0]] ;

```
  #end
  TrExpr[[ E1, {E2 … En}, V, e1 ]] ; … ; TrExpr[[ En, {}, V, en ]] ;
  #if (m is a static method)
    call m(e1 … en) ;
  #else
    call m(e0 e1 … en) ;
  #end
  Protect[[ RES, p, V, r ]]
 #end
```

```
TrExpr[[ super . m (E1 E2 … En) , p, V, r ]] ==
 #var e1 … en in
  TrExpr[[ E1, {E2 … En}, V, e1 ]] ; … ; TrExpr[[ En, {}, V, en ]] ;
  #if (m is a static method)
    call m(e1 … en) ;
  #else
    call m(this e1 … en) ;
  #end
  Protect[[ RES, p, V, r ]]
 #end
```

```
TrExpr[[ new T m (E1 E2 … En), p, V, r ]] ==
 #var e1 … en in
  TrExpr[[ E1, {E2 … En}, V, e1 ]] ; … ; TrExpr[[ En, {}, V, en ]] ;
  call m(e1 … en) ;
  assume typeof(RES) == T ;
  Protect[[ RES, p, V, r ]]
 #end
```

5 Translating specification expressions

This section describes a function *TrSpecExpr* that translates a specification expression into a guarded command expression. Recall from ESCJ 17, *ESC/Java Annotation Reference Manual*, specification expressions are similar to Java expressions, but they are *pure* (that is, they are side-effect free), they cannot raise exceptions, they are *total* (that is, their evaluation cannot "go wrong"), and they may include additional constructs such as quantifiers and *PRE* and *fresh*. Guarded command expressions are similar to specification expressions in that they are pure and total and may include quantifiers. However, guarded command expressions do not include *PRE* and *fresh*, for example.

Function *TrSpecExpr*[[E, sp, st]], where E is an specification expression and sp and st are domain-disjoint partial maps from variables to guarded command expressions, returns a guarded command expression corresponding to E, in which every occurrence of a variable v in E and in *Domain*[[sp]] has been replaced by sp[[v]], and every occurrence of a variable v in a *PRE* expression in E and in *Domain* [[st]] has been replaced by st[[v]]. We require that if E contains a *fresh* expression, then *alloc* is in *Domain* [[st]].

In other parts this document, as a notational convenience, we write *TrSpecExpr*[[E]] for *TrSpecExpr*[[E, {}, {}]]. Here, as a notational convenience, we abuse the notation sp[[v]] to mean

```
#if (v in Domain[[sp]])
  sp[[v]]
#else
  v
#end
```

and similarly for st.

```
TrSpecExpr[[ this, sp, st ]] ==
  sp[[this]]
```

TrSpecExpr[[C, sp, st]] == /* where C is a literal */
C

TrSpecExpr[[v, sp, st]] == /* where v is a local variable, parameter, static field, *RES*, or *LS* */
sp[[v]]

TrSpecExpr[[E.g, sp, st]] == /* where g is a static field */
sp[[g]]

TrSpecExpr[[E.f, sp, st]] == /* where f is an instance variable */
sp[[f]] [*TrSpecExpr*[[E, sp, st]]]

TrSpecExpr[[E0[E1], sp, st]] ==
sp[[*elems*]] [*TrSpecExpr*[[E0, sp, st]]] [*TrSpecExpr*[[E1, sp, st]]]

TrSpecExpr[[E[*], sp, st]] ==
sp[[*elems*]] [*TrSpecExpr*[[E, sp, st]]]

TrSpecExpr[[unOp E, sp, st]] == /* where unOp is a unary operator, possibly *typeof* or *elemtype* or *min* */
unOp(*TrSpecExpr*[[E, sp, st]])

TrSpecExpr[[E0 binOp E1, sp, st]] == /* where binOp is a binary operator, possibly && or || or ==> or <: */
binOp(*TrSpecExpr*[[E0, sp, st]], *TrSpecExpr*[[E1, sp, st]])
Here, we are using prefix notation for applications of all binary operators. Elsewhere in this document, we frequently use infix notation.

TrSpecExpr[[G ? E0 : E1, sp, st]] ==
term$conditional(*TrSpecExpr*[[G, sp, st]], *TrSpecExpr*[[E0, sp, st]], *TrSpecExpr*[[E1, sp, st]])

TrSpecExpr[[E **instanceof** T, sp, st]] ==
is(*TrSpecExpr*[[E, sp, st]], *TrType*[[T]])

TrSpecExpr[[(T) E, sp, st]] ==
cast(*TrSpecExpr*[[E, sp, st]], *TrType*[[T]])

TrType[[T]] == /* where T is a primitive type or declared type */
T

TrType[[T[]]] ==
array(*TrType*[[T]])

TrSpecExpr[[(**forall** (T1 x1) … (Tn xn) E), sp, st]] ==
/* where we require the domains of sp and st to be disjoint from {x1, …, xn} */
We believe our translation never violates this requirement, but it might be worthwhile to include a runtime check in the translator.
(**forall** x1 … xn :: *TypeCorrectAs*[[x1, T1]] && … && *TypeCorrectAs*[[xn, Tn]]
==> *TrSpecExpr*[[E, sp, st]])
We should also replace occurrences of *alloc* in *TypeCorrectAs*[[x1, T1]] && … && *TypeCorrectAs*[[xn, Tn]] with sp[[*alloc*]].

TypeCorrectAs[[v, T]] ==
TypeAndNonnullCorrectAs[[v, T, *false*]]

TypeAndNonnullCorrectAs[[v, T, isNonNull]] ==
is(v, T)
#**if** (T is a reference type)
&& *allocTime*(v) < *alloc*

#**if** (isNonNull)
&& v != *null*
#**end**
#**end**

An optimization would be to generate *typeof*(v) <: T instead of *is*(v, T) if T is a reference type and isNonNull is *true*.

TrSpecExpr[[(**exists** (T1 x1) … (Tn xn) E), sp, st]] ==
/* where we require the domains of sp and st to be disjoint from {x1, …, xn} */

We believe our translation never violates this requirement, but it might be worthwhile to include a runtime check in the translator.

(**exists** x1 … xn :: *TypeCorrectAs*[[x1, T1]] && … && *TypeCorrectAs*[[xn, Tn]]
&& *TrSpecExpr*[[E, sp, st]])

We should also replace occurrences of *alloc* in *TypeCorrectAs*[[x1, T1]] && … && *TypeCorrectAs*[[xn, Tn]] with sp[[*alloc*]].

TrSpecExpr[[(**lblpos** L E), sp, st]] ==
(**lblpos** L *TrSpecExpr*[[E, sp, st]])

TrSpecExpr[[(**lblneg** L E), sp, st]] ==
(**lblneg** L *TrSpecExpr*[[E, sp, st]])

TrSpecExpr[[*PRE*(E), sp, st]] ==
TrSpecExpr[[E, sp ∪ st, {}]]

It is okay to pass the empty map as the third parameter because our annotation language forbids uses of *PRE* or *fresh* within an argument of *PRE*.

TrSpecExpr[[*fresh*(E), sp, st]] ==
TrSpecExpr[[E, sp, st]] != *null* && st[[*alloc*]] < *allocTime*(*TrSpecExpr*[[E, sp, st]])

We omit the requirement *allocTime*(*TrSpecExpr*[[E, sp, st]]) < *alloc* because this condition is introduced by other mechanisms when it is needed.

6 Translating statements

TrStmt[[S, V]], where S is a Java statement and V is a set of temporary variable names, translates S into a guarded command. Temporary variables used in that command can either be local to the command or added to the in-out parameter V. We assume again that variables introduced in translation are fresh.

TrStmt[[**block** S1 … Sn **end**, V]] ==
var x1 … xk init$xi … init$xj **in**
TrStmt[[S1, V]] ; … ; *TrStmt*[[Sn, V]]
end

where x1 … xk are the variables introduced by those of the statements S1 … Sn that are Java **var** statements, and xi … xj are the (not necessarily contiguous) subset of x1 … xk that are declared as **uninitialized**.

TrStmt[[**var** M1 … Mn x, V]] ==
skip

TrStmt[[**var** M1 … Mn x = E, V]] ==
#**if** (x declared with **uninitialized**)
Assign[[x, E, V]]
#**else**
Eval[[x = E, V]]
#**end**

where

Assign[[x, E, V]] ==

#**var** e **in**
TrExpr[[E, {}, V, e]] ;
x = e ;
#**end**

and where

Eval[[E, V]] ==
#**var** junk **in**
TrExpr[[E, {}, V, junk]]
#**end**

Note that, if x is declared as **uninitialized**, then *Eval*[[x = E, V]] sets init$x to *true*, whereas *Assign*[[x, E, V]] does not assign to init$x (except perhaps as a side effect of evaluating E).

TrStmt[[**label** L S, V]] ==
block L: *TrStmt*[[S, V]] **end**

TrStmt[[**skip**, V]] ==
skip

TrStmt[[**eval** E, V]] ==
Eval[[E, V]]

TrStmt[[**if** (E) S0 **else** S1, V]] ==
#**var** e **in**
TrExpr[[E, {}, V, e]] ;
if e **then** *TrStmt*[[S0]] **else** *TrStmt*[[S1]] **end**
#**end**

TrStmt[[**break** L, V]] ==
EC = L ; **raise**

TrStmt[[**continue** L, V]] ==
EC = continue$L ; **raise**

TrStmt[[**return**, V]] ==
EC = *ec$return* ; **raise**

TrStmt[[**return** E, V]] ==
Assign[[*RES*, E, V]] ;
EC = *ec$return* ; **raise**

TrStmt[[**throw** E, V]] ==
Assign[[*XRES*, E]] ;
check *NullPointerException* : *XRES* != *null* ;
EC = *ec$throw* ; **raise**

We perform the *XRES* != *null* check here, and so does Sun's Java implementation, but it is not documented in either *JLS* or the Java bytecode specification.
Although Sun's Java implementation turns throwing *null* into a *NullPointerException*, we could actually give this error a different name that would better describe the error.

TrStmt[[**try** S **catch** (T1 x1 S1) (T2 x2 S2) … (Tn xn Sn) **end**, V]] ==
TrStmt[[S, V]] !
if *EC* != *ec$throw* **then skip else**
if *typeof*(X*RES*) <: T1 **then var** x1 **in assume** x1 == X*RES* ; *TrStmt*[[S1, V]] **end else**
if *typeof*(X*RES*) <: T2 **then var** x2 **in assume** x2 == X*RES* ; *TrStmt*[[S2, V]] **end else**
…
if *typeof*(X*RES*) <: Tn **then var** xn **in assume** xn == X*RES* ; *TrStmt*[[Sn, V]] **end else**

```
      raise
    end
   …
  end
 end
end
```

TrStmt[[**try** S0 **finally** S1, V]] ==
 #**var** C0, C1 **in**
  #C0 = *TrStmt*[[S0, V]] ;
  #V = V *ec res xres* ;
  #C1 = *TrStmt*[[S1, V]] ;
  (C0 !
   assume *ec* == *EC* && *res* == *RES* && *xres* == *XRES* ;
   C1 ;
   EC = *ec* ; *RES* = *res* ; *XRES* = *xres* ; **raise**
) ; C1
 #**end**

TrStmt[[L: **switch** (E) (**case** [E1] S11 … S1n₁) … (**case** [Ek] Sk1 … Skn$_k$) **end**, V]] ==
 var x1 … xk init$xi … init$xj **in**
  #V = V *e* ;
  Assign[[*e*, E, V]] ;
  block L:
   ((((… ((**assume** C1
                 ; *TrStmt*[[S11, V]] ; … ; *TrStmt*[[S1n$_1$, V]]
)
              [] **assume** C2
)
              ; *TrStmt*[[S21, V]] ; … ; *TrStmt*[[S2n$_2$, V]]
           …
)
       [] **assume** C(k-1)
)
       ; *TrStmt*[[S(k-1)1, V]] ; … ; *TrStmt*[[S(k-1)n$_{k-1}$, V]]
)
   [] **assume** Ck
)
   ; *TrStmt*[[Sk1, V]] ; … ; *TrStmt*[[Skn$_k$, V]]
  end
 end

where x1 … xk are the variables introduced by those of the statements S11 … Skn$_k$ that are Java **var** statements, xi … xj are the (not necessarily contiguous) subset of x1 … xk that are declared as **uninitialized**, and Ci is *e* == *TrSpecExpr*[[Ei]] if Ei is mentioned, or
  e != *TrSpecExpr*[[E1]] && … && *e* != *TrSpecExpr*[[E(i-1)]] &&
  e != *TrSpecExpr*[[E(i+1)]] && … && *e* != *TrSpecExpr*[[Ek]]
if Ei is omitted.

This translation of the **switch** statement relies on the assumption that all the case labels Ei are constant expressions that evaluate to distinct values, just like the Java language specification requires [*JLS*, 14.9].

TrStmt[[**synchronized** (E) S, V]] ==
 #V = V *mu* ;
 Assign[[*mu*, E, V]] ;
 check *LockingOrderViolation* : *mutex$atmost*(*max*(*LS*), *mu*) || *LS*[*mu*] ;
  We could introduce an annotation or command-line switch to drop the second disjunct, thus disallowing reentrant locking.
 TrSynchronizedBody[[*mu*, S, V]]

where

TrSynchronizedBody[[mu, S, V]] ==
 #V = V *newLS* ;
 assume (*mutex$atmost*(*max*(*LS*), mu) && mu == *max*(*newLS*)) ||
   (*LS*[mu] && *newLS* == *LS*) ;
 assume *newLS* == *store*(*LS*, mu, *bool$true*) ;
 assume *newLS* == *asLockSet*(*newLS*) ;
 (*TrStmt*[[S, V]])[*LS* *newLS*]

An alternative translation of the **synchronized** statement would be:

TrStmt[[**synchronized** (E) S, V]] ==
 … as before until after the **assume** command …
 #V = V *oldLS* ;
 assume *oldLS* = *LS* ;
 LS = *newLS* ;
 TrStmt[[**try** S **finally** *LS* = *oldLS*, V]]

However, with this alternative translation, there is a risk that the prover will need to do a case analysis on normal versus exceptional termination of S in order to establish that the value of *LS* is the same after the **synchronized** statement as before.

The substitution in the actual translation can alternatively be implemented by passing a locking set variable name as a new parameter of *TrStmt* and of *TrExpr*, emitting this new parameter where the translation now emits *LS*. As another alternative, the current locking set name could be kept in a global variable.

TrConstructorCallStmt[[**construct** m (E1 E2 … En), T, V]] ==
 #**var** e1 … en **in**
  TrExpr[[E1, {E2 … En}, V, e1]] ; … ; *TrExpr*[[En, {}, V, en]] ;
  call m(e1 … en) ;
  this = *RES*
 #**end**

This is the only place where *this* is assigned.

TrStmt[[**assert** SE, V]] ==
 check *AssertionViolation* : *TrSpecExpr*[[SE]]

TrStmt[[**assume** SE, V]] ==
 assume *TrSpecExpr*[[SE]]

TrStmt[[**unreachable**, V]] ==
 check *ReachabilityViolation* : *false*

6.0 Translating loops

This section defines *TrStmt* on loops in terms of a function *MakeLoop*, which is also defined in this section.

TrStmt[[L: **while** (G) { **loop_invariant** J1 … Jn } S, V]] ==
 #**var** W = {}, CG = *Guard*[[G, W]], CS = *TrStmt*[[S, W]] **in**
  MakeLoop[[**var** W **in** CG ; **block** continue$L : CS **end end**, J1 … Jn, L, V]]
 #**end**

TrStmt[[L: **do** { **loop_invariant** J1 … Jn } S **while** (G), V]] ==
 #**var** W = {}, CS = *TrStmt*[[S, W]], CG = *Guard*[[G, W]] **in**
  MakeLoop[[**var** W **in block** continue$L : CS **end** ; CG **end**, J1 … Jn, L, V]]
 #**end**

TrStmt[[L: **for** (**var** M1 … Mn x [= E] ; G ; E1 … En) { **loop_invariant** J1 … Jn } S, V]] ==

Out of curiosity, does our annotation language allow one of x's modifiers to be **uninitialized**?

 #**var** W = {}, CG = *Guard*[[G, W]], CS = *TrStmt*[[S, W]],
   CE = (*Eval*[[E1, W]] ; … ; *Eval*[[En, W]]) **in**

```
    var x in
      TrStmt[[ var M1 … Mn x [= E], V ]] ;
      MakeLoop[[ var W in CG ; block continue$L :  CS end ; CE end, J1 … Jn, L, V ]]
    end
  #end
```

```
TrStmt[[ L: for (F1 … Fk ; G ; E1 … En) { loop_invariant J1 … Jn } S, V ]] ==
  #var W = {}, CG = Guard[[ G, W ]], CS = TrStmt[[ S, W ]],
            CE = ( Eval[[ E1, W ]] ; … ; Eval[[ En, W ]] ) in
    Eval[[ F1, V ]] ; … ; Eval[[ Fk, V ]] ;
    MakeLoop[[ var W in CG ; block continue$L :  CS end ; CE end, J1 … Jn, L, V ]]
  #end
```

We now define *MakeLoop*. For any guarded command Body, two-state specification expressions J1 … Jn denoting invariants, label L, and any V,

```
MakeLoop[[ Body, J1 … Jn, L, V ]] ==
  #var LoopTargs = NTargets[[ Body, {} ]], wt = MakeSubst[[ LoopTargs, L ]] in
    block L:
      var wt[[ LoopTargs ]] in
        #for w in LoopTargs do
          assume wt[[ w ]] == w ;
```

Should this be an assumption or an assignment? The choice may have performance implications.

```
        #end
        loop
          { inv LoopInvariantViolation :  TrSpecExpr[[ J1, {}, wt ]]
                …
                LoopInvariantViolation :  TrSpecExpr[[ Jn, {}, wt ]]
                LoopObjectInvariants[[ LoopTargs ]] }
          #for w in LoopTargs do
            TargetTypeCorrect[[ w, wt ]] ;
          #end
          Body
        end
      end
    end
  #end
```

If we use *DesugarLoopFast* (section 2.1), then the commands generated by the calls to *TargetTypeCorrect* (all of which are **assume** commands) are redundant and can be omitted.
It may sometimes be desirable to leave out the loop invariants generated by the call to *LoopObjectInvariants*. This could be under the control of an ESC/Java command-line switch, but we conjecture that most ESC/Java users would want to omit the call to *LoopObjectInvariants* precisely when *DesugarLoopFast* (section 2.1) is used.

where, for any list of variables v1 … vm and location L, *MakeSubst* is defined as follows:

```
MakeSubst[[ v1 … vm, L ]] ==
  {v1   v1@L, …, vm   vm@L}
```

MakeSubst allocates the AST nodes for the adorned names.

and where, for any list of variables LoopTargs, *LoopObjectInvariants* is defined as follows:

```
LoopObjectInvariants[[ LoopTargs ]] ==
  #for every static invariant J in scope, whose free variables intersect with LoopTargs
    ObjectInvariantViolationForLoop :  TrSpecExpr[[ J ]]
  #end
  #for every object invariant J in scope, whose free variables intersect with LoopTargs
    ObjectInvariantViolationForLoop :  (ALL this :: this != null ==> TrSpecExpr[[ J ]] )
  #end
```

and where, for any Java expression G, *Guard* is defined as follows:

Guard[[G, V]] ==
  #**var** g **in**
    TrExpr[[G, {}, V, g]] ;
    if g **then skip else raise** L **end**
  #**end**

and where, for any variable w and variable map wt, *TargetTypeCorrect* is defined as follows:

TargetTypeCorrect[[w, wt]] ==
  #**if** (w is a local variable or static field)
    assume *TypeCorrect*[[w]]
  #**elsif** (w is an instance variable)
    assume *FieldTypeCorrect*[[w]] ;
  #**elsif** (w is *elems*)
    assume *ElemsTypeCorrect*[[w]] ;
  #**elsif** (w is *alloc*)
    assume wt[[*alloc*]] < *alloc* ;
  #**elsif** (w is an *init$* variable)
    assume wt[[w]] ==> w
  #**end**

TypeCorrect[[v]] ==
  #**let** T be the declared type of v **in**
    #**if** (v is declared with **non_null**)
      TypeAndNonnullCorrectAs[[v, T, *true*]]
    #**else**
      TypeAndNonnullCorrectAs[[v, T, *false*]]
    #**end**
  #**end**

FieldTypeCorrect[[f]] ==
  #**let** T be the declared type of f **in**
    f == *asField*(f, T)
    #**if** (T is a reference type)
      && *fClosedTime*(f) < *alloc*
      #**if** (v is declared with **non_null**)
        && (ALL *s* :: *allocTime*(*s*) < *alloc* ==> f[*s*] != *null*)
      Do we need the antecedent?
      We have intentionally omitted the conjuncts *s* != *null* && *is*(*s*, T) from the antecedent, because we think they are not needed.
      We had better check that we haven't done some other simplification elsewhere that would require f[*null*] == *null*.
      #**end**
    #**end**
  #**end**

ElemsTypeCorrect[[e]] ==
  e == *asElems*(e) && *eClosedTime*(e) < *alloc*

7 Synthesizing method specifications

In this section, we explain how to synthesize a method specification from an annotated Java method declaration and a scope. In particular, for a method m, a scope Sc, and list of variables SynTargs (called syntactic targets), we define two functions *GetSpecForCall*[[m, Sc]] and *GetSpecForBody*[[m, Sc, SynTargs]], each of which returns a method specification of the form described in section 2.1:

    spec T m(p1 … pn) **throws** {X1 … Xx}
    precondition P1 … **precondition** Pj
    targets D1 … Dk

whole-targets wt
postcondition Q1 … **postcondition** Qm

The two functions are defined as follows:

GetSpecForCall[[m, Sc]] ==
ExtendSpecForCall[[*GetCommonSpec*[[m, Sc]], Sc]]

GetSpecForBody[[m, Sc, SynTargs]] ==
ExtendSpecForBody[[*GetCommonSpec*[[m, Sc]], Sc, SynTargs]]

SynTargs will in fact always be the syntactic targets of the body of the method, see section 8.1.
The fact that *ExtendSpecForBody* takes a list of syntactic targets as a parameter may seem a little odd: Is the meaning of the method specification influenced by the implementation of the method? The reason for this parameter is so that *ExtendSpecForBody* can reduce the number of checked postconditions it adds, by suppressing those that are tautologies in the light of the body.

GetCommonSpec[[m, Sc]] ==
TrMethodDecl[[*FilterMethodDecl*[[*GetCombinedMethodDecl*[[m]], Sc]]]]

where *GetCombinedMethodDecl*, *FilterMethodDecl*, *TrMethodDecl*, *ExtendSpecForCall*, and *ExtendSpecForBody* are defined below.

The function *GetCombinedMethodDecl* combines the declaration of m with the declarations of the methods that m overrides, producing a method declaration of the form:

method T m(p1 … pn) **throws** {X1 … Xx}
requires P1 … **requires** Pk
modifies w1 … wu
ensures Q1 … **ensures** Qh

In particular, *GetCombinedMethodDecl* is responsible for:

- Assembling the signature T m(p1 … pn) **throws** {X1 … Xx}. In the case of an instance method, this process includes prepending *this* to the list of declared parameters. In the case of a constructor, the return type T is the class containing the constructor declaration. We assume that m is represented in a form from which one can extract whether or not m is a constructor, and that the declared parameters are represented in a form from which one can extract information such as type information and **non_null** information.
- Combining the **requires** pragmas of the method declaration of m or of a method that m overrides.
- Combining the **modifies** and **also_modifies** pragmas of the method declaration of m and of the methods that m overrides.
- Combining the **ensures** and **also_ensures** pragmas of the method declaration of m and of the methods that m overrides.
- Replacing occurrences of the parameter names in the specifications combined from overridden methods with the corresponding parameter names of the overriding method.

Note that all expressions in the **requires**, **modifies**, and **ensures** clauses of the declaration returned by *GetCombinedMethodDecl* are specification expressions, not guarded command expressions.

Function *FilterMethodDecl* prunes away parts of the method declaration that mention variables that are not in scope. In particular, *FilterMethodDecl* is responsible for:

- Filtering the **modifies** list, removing designators that mention variables not in scope.
  This is unsound, but seems necessary in the absence of abstraction in the annotation language.
- Removing postconditions that mention variables not in scope.
  This is sound provided that the scope of each implementation gives rise to no pruning. The ESC/Java front end produces a syntactic warning if a programmer mentions a private variable in the specification of a non-private, non-final method (or override) in a non-final class. All other cases are sound.

Note that non-public classes mentioned as types of parameters, or as exceptions in the throws set, of a public method are not filtered out, despite the fact that a public caller do not have access to these classes. Courtesy of Java, thank you.

Function *TrMethodDecl* translates a (combined) method declaration into a method specification (see section 2.1). Given a method declaration, *TrMethodDecl* is responsible for:

- Generating checked preconditions from **non_null** parameter annotations.
- Translating **requires** clauses into checked preconditions.
- Generating checked preconditions for **synchronized** methods.
- Translating the **modifies** clause into a list of target designators, and adding *alloc* to this list. (This translation includes changing the specification designator E.g into just g, when g is a static field. Note that in the annotated Java AST, the name g has already been disambiguated, so E is not needed for the disambiguation).

In the presence of data groups, *TrMethodDecl* would be a nice place to compute downward closures.

- Computing a whole-targets map from the target designators.
- Translating **ensures** clauses into checked postconditions.
- Generating free preconditions from the types of the parameters, stating the type correctness of the parameters.
- Generating a free postcondition from the result type, stating the type correctness of the result.
- Generating a checked postcondition from the **throws** set, stating which exceptions, if any, are acceptable outcomes of the method.
- Generating a free postcondition from the **throws** set, stating the type correctness of any thrown exception.
- Generating free postconditions from the whole targets, stating their type correctness.
- Generating free postconditions from **non_null** annotations of the whole targets.

ExtendSpecForCall and *ExtendSpecForBody* extend what *TrMethodDecl* produces to take into account object invariants. This is done differently for callers and callees, so there are two functions. Function *ExtendSpecForCall* is responsible for:

- Generating checked preconditions from heuristically chosen object invariants and static invariants.
- Generating postconditions from the whole targets and from the object invariants and static invariants in scope, stating that the call does not invalidate any of the invariants.

and function *ExtendSpecForBody* is responsible for:

- Generating preconditions from the object invariants and static invariants in scope.
- Generating checked postconditions from whole targets, syntactic targets, object invariants, and static invariants, stating that the invariants are maintained.

7.0 GetCombinedMethodDecl

This section describes, for a given method name m, the various components of the result of *GetCombinedMethodDecl*[[m]].

7.0.0 Signature

If m is a static method declared with parameters p1 … pn and result type T (possibly **void**) and throws set {X1 … Xx}, or if m is a constructor of a class T with declared parameters p1 … pn and throws set {X1 … Xx}, then *GetCombinedMethodDecl*[[m]] returns the signature

T m(p1 … pn) **throws** {X1 … Xx}

If m is an instance method declared with parameters p1 … pn and result type T (possibly **void**) and throws set {X1 … Xx}, then *GetCombinedMethodDecl*[[m]] returns the signature

T m(*this* p1 … pn) **throws** {X1 … Xx}

Note that the variables p1 … pn in these cases above have been unique-ified by the parser, which creates a distinct AST node for each variable, field, or parameter declaration. In particular, the declared parameters of a method are distinct from the declared parameters of any method that it overrides, even if the same textual names are used.

We define the *reference declaration* of a method m as follows: If m is not an override, the reference declaration of m is the declaration of m; if m overrides a method m´ in a superclass, the reference declaration of m is the reference declaration of m´. The reference declaration of a constructor is the constructor itself; constructors cannot be overridden, since the constructor name declared must be the name of the class in which it occurs [*JLS*, 8.6].

In the signatures described above, a parameter is considered to be declared **non_null** if the reference declaration of m declared the corresponding parameter as **non_null**. Since the ESC/Java annotation language allows the **non_null** pragma to be used only for the parameters of reference declarations, the only way a parameter of an overriding method can be **non_null** is by inheritance of the **non_null** attribute as just described.

For generating location information in verification conditions, we need a mechanism by which given a parameter, one can extract the location of any inherited **non_null** pragma.

For use in the rest of this section, we now define a function that returns a substitution map to the parameter names of a method m from the parameter names of the methods that m overrides. For any method or constructor m:

ParameterMappings[[m]] ==
 #**if** (m is a reference declaration)
  { }
 #**elsif** (m directly overrides a method m´)
  #**let** p1 … pn be the declared parameters of m **in**
   ParmeterMappingsAux[[m´, p1 … pn]]
  #**end**
 #**end**

where

ParmeterMappingsAux[[m, p1 … pn]] ==
 #**let** q1 … qn be the declared parameters of m **in**
  #**if** (m is a reference declaration)
   {q1 p1, …, qn pn}
  #**elsif** (m directly overrides a method m´)
   {q1 p1, …, qn pn} ∪ *ParmeterMappingsAux*[[m´, p1 … pn]]
  #**end**
 #**end**

In the rest of this section, let pmap denote *ParameterMappings*[[m]].

7.0.1 Combining requires clauses

Suppose the reference declaration of m is declared with the **requires** pragmas:

requires P1
…
requires Pk

We should state the restriction that all variables mentioned in a **requires** pragma must be as visible as the method it specifies. Furthermore, if m is a constructor, then its **requires** pragmas are not allowed to mention **this**, either implicitly or explicitly.

Then, *GetCombinedMethodDecl*[[m]] includes the following **requires** clause:

requires pmap[[P1]]
…
requires pmap[[Pk]]

7.0.2 Combining modifies lists

Suppose

modifies w1 … w..
…
modifies w.. … w..
also_modifies w.. … w..
…
also_modifies w.. … wu

are the **modifies** and **also_modifies** pragmas of m and the methods it transitively overrides. (Note that a constructor is never annotated with an **also_modifies** pragma, because a constructor cannot be overridden.) *GetCombinedMethodDecl*[[m]] then includes

modifies pmap[[w1]] … pmap[[wu]]

7.0.3 Combining ensures clauses

Suppose

ensures Q1
…
ensures Q..
also_ensures Q..
…
also_ensures Qh

are the **ensures** and **also_ensures** pragmas of m and the methods it transitively overrides. (Note that a constructor is never annotated with an **also_ensures** pragma, because a constructor cannot be overridden.) *GetCombinedMethodDecl*[[m]] then includes

ensures pmap[[Q1]]
…
ensures pmap[[Qh]]

7.1 FilterMethodDecl

Given a method declaration decl of the form

method T m(p1 … pn) **throws** {X1 … Xx}
requires P1 … **requires** Pk
modifies w1 … wu
ensures Q1 … **ensures** Qh

and a scope Sc, we define:

FilterMethodDecl[[decl, Sc]] ==
method T m(p1 … pn) **throws** {X1 … Xx}
requires P1 … **requires** Pk
modifies
#**for** w **in** w1 … wu **do**
#**if** (all variables in w are visible in Sc)
w
#**end**

```
#end
#for Q in Q1 … Qh do
  #if (all variables in Q are visible in Sc)
        ensures Q
  #end
#end
```

Note that for a static field g, a specification expression E.g would always evaluate to the same value as g. Function *FilterMethodDecl*, as defined here, filters out specification designators and postconditions containing expressions of the form E.g whenever E contains some variable not in scope, even if g is a static field that is in scope. An alternative design would be to transform E.g to g before filtering. In the current design, the translation of E.g into g occurs in *TrMethodDecl*.

7.2 TrMethodDecl

This section describes the various components of the result of *TrMethodDecl* for a given method declaration

> **method** T m(p1 … pn) **throws** {X1 … Xx}
> **requires** P1 … **requires** Pk
> **modifies** w1 … wu
> **ensures** Q1 … **ensures** Qh

The signature returned by *TrMethodDecl* is the same as the one given.

7.2.0 Preconditions

We now describe the list of **precondition** clauses that the *TrMethodDecl* function returns.

TrMethodDecl[[m]] includes

```
#for p in p1 … pn do
  #if (p is this)
    #let U be the class that declares m in
        precondition Free :  is(this, U) && allocTime(this) < alloc
        precondition NullPointerException :  this != null
    #end
  #else
    #let U be the type of p in
        precondition Free :  is(p, U)
      #if (U is a reference type)
        precondition Free :  allocTime(p) < alloc
        #if (p is declared as non_null)
```

Recall that p is considered to be declared as **non_null** if the corresponding parameter in the reference declaration of m is declared as **non_null**.

```
        precondition NonNullViolation :  p != null
        #end
      #end
    #end
  #end
#end
```

TrMethodDecl[[m]] also includes

> **precondition** *PreconditionViolation* : *TrSpecExpr*[[P1]]
> …
> **precondition** *PreconditionViolation* : *TrSpecExpr*[[Ph]]

Finally, if m is a **synchronized** instance method, then *TrMethodDecl*[[m]] includes

> **precondition** *LockingOrderViolation* : *mutex$atmost*(*max*(*LS*), *this*) || *LS*[*this*]

We could introduce an annotation or command-line switch to drop the second disjunct, thus disallowing reentrant locking. For now, to forbid reentrancy into a **synchronized** method, the programmer must supply an explicit **requires** clause.

and if m is a **synchronized** static method of a class U, then *TrMethodDecl*[[m]] includes

> **precondition** *LockingOrderViolation* : *mutex$atmost*(*max*(*LS*), U) || *LS*[U]

In the second case, U is the class object [*JLS*, 17.13 and 20.3].
Currently, the annotation language does not let a user mention U as an argument to <, <=, or *LS*[ ], so there is no way to discharge proof obligations relating to the position of class objects in the locking order.

7.2.1 Targets

We now describe the **targets** and **whole-targets** clauses that the *TrMethodDecl* function returns.

TrMethodDecl[[m]] includes

> **targets** *BasicTargets*[[w1 … wu]]

where function *BasicTargets* is defined as:

BasicTargets[[w1 … wu]] ==
 TrSpecExpr[[w1]] … *TrSpecExpr*[[wu]] *alloc*

Corresponding to the designator targets, *TrMethodDecl*[[m]] also includes the following whole targets map:

> **whole-targets** *MakeSubst*[[*ShaveAll*[[*BasicTargets*[[w1 … wu]]]], pre]]

We assume that *MakeSubst* creates AST nodes for the new names.

where *ShaveAll* is defined to be a duplicate-free list of variable names, as follows:

ShaveAll[[D1 … Dd]] ==
 #**for** D **in** D1 … Dd **do**
  Shave[[D]]
 #**end**

but with duplicates removed, and *Shave* is defined as follows: for any variable v and expressions E0 and E1,

- *Shave*[[v]] == v
- *Shave*[[v[E0]]] == v
- *Shave*[[v[E0][E1]]] == v

7.2.2 Postconditions

We now describe the list of **postcondition** clauses that the *TrMethodDecl* function returns. Throughout this section, we let wt denote the map created for the **whole-targets** clause as described above.

Every method and constructor body is allowed to allocate new objects, and hence may advance the current allocation time. Thus, *TrMethodDecl*[[m]] includes

> **postcondition** *Free* : wt[[*alloc*]] < *alloc*

Note that if our translation were to assume free postconditions at the end of a body, as a possible aid in discharging the checked postconditions, the free postcondition described here may provide more aid than warranted. The problem is that the body might do no allocations, in which case wt[[*alloc*]] == *alloc* at the end of the body. Were this to become an issue, we could change the < in this free postcondition to an <=. For now, we're leaving it as <, because we currently don't assume free postconditions at the end of the body and we don't know if using <= would give rise to case splits in reasoning about calls.

This postcondition is free, because the programming language offers no way to decrease the allocation time.

If m is a constructor, then *TrMethodDecl*[[m]] includes

> **postcondition** *Free* : *RES* != *null* && wt[[*alloc*]] < *allocTime*(*RES*)

If T is not **void**, then *TrMethodDecl*[[m]] includes

> **postcondition** *Free* : *TypeCorrectAs*[[*RES*, T]]

Note that no antecedent *EC* == *ec$return* is needed, because only if the call returns normally does the caller actually use *RES*.

TrMethodDecl[[m]] also includes

> **postcondition** *Free* : *EC* == *ec$throw* ==>
> *XRES* != *null* && *typeof*(*XRES*) <: *Throwable* && *allocTime*(*XRES*) < *alloc*
> **postcondition** *UnexpectedException* :
> *EC* == *ec$return* ||
> (*EC* == *ec$throw* && (*typeof*(*XRES*) <: X1 || … || *typeof*(*XRES*) <: Xx))

If the throws set is empty, then this checked postcondition simplifies to *EC* == *ec$return*.

Finally, *TrMethodDecl*[[m]] includes

#**for** Q **in** Q1 .. Qh **do**
> **postcondition** *PostconditionViolation* : *EC* == *ec$return* ==> *TrSpecExpr*[[Q, {}, wt]]

If the throws set is empty, then the antecedent *EC* == *ec$return* can be dropped.

#**end**

7.3 ExtendSpecForCall

This section describes, for a given method specification spec of the form

> **spec** T m(p1 … pn) **throws** {X1 … Xx}
> **precondition** P1 … **precondition** Pj
> **targets** D1 … Dk
> **whole-targets** wt
> **postcondition** Q1 … **postcondition** Qm

and a scope Sc, the various components of the result of *ExtendSpecForCall*[[spec, Sc]]. Function *ExtendSpecForCall* returns a method specification like spec but extended with additional **precondition** and **postcondition** clauses. These conditions arise from heuristically chosen object invariants and static invariants.

7.3.0 Adding preconditions

We now describe the list of additional **precondition** clauses that *ExtendSpecForCall* returns.

We start with a couple of definitions. An invariant J declared in a class T is an *object invariant* of T if J mentions **this**, and is a *static invariant* of T otherwise. An invariant is called Sc-visible if it is in scope in Sc.

These definitions would be better placed elsewhere, perhaps near the (to be written) AST grammar of declarations.

For every static invariant J in scope Sc, *ExtendSpecForCall*[[spec, Sc]] includes

> **precondition** *StaticInvariantViolation* : *TrSpecExpr*[[J]]

For each static field g in scope Sc, if the static type of g is a class U, then *ExtendSpecForCall*[[spec, Sc]] includes

> **precondition** *ObjectInvariantViolation* : g == *null* || *TrSpecExpr*[[J, {*this*   g}, {}]]

The first disjunct can be suppressed if g is declared as **non_null**.

for every Sc-visible object invariant J of any superclass of U.

For each parameter p in the signature of spec, if the static type of p is a class U (or if p is *this* and m is a method of a class U), then *ExtendSpecForCall*[[spec, Sc]] includes

> **precondition** *ObjectInvariantViolation* : p == *null* || *TrSpecExpr*[[J, {*this*   p}, {}]]

The first disjunct can be suppressed if p is *this* or is declared as **non_null**.

for every Sc-visible object invariant J of any superclass of U.

7.3.1 Adding postconditions

We now describe the list of additional **postcondition** clauses that *ExtendSpecForCall* returns.

The postconditions generated here are used only in the desugaring of calls. In this context, the predicates of all postconditions are assumed and the error names are ignored. Since the error names are ignored, we have written them as *Free*.

For every Sc-visible static invariant J, *ExtendSpecForCall*[[spec, Sc]] includes

> **postcondition** *Free* : *TrSpecExpr*[[J]]

As an important optimization, this **postcondition** clause is suppressed for J if the free variables of J are disjoint from the domain of wt (in which case this postcondition is a tautology).

For every Sc-visible object invariant J of any class U, *ExtendSpecForCall*[[spec, Sc]] includes

#**if** (m is a constructor, and U is a proper subtype of T)
> **postcondition** *Free* : (ALL *s* :: *TypeCorrectAs*[[*s*, U]] && *s* != *null* && *s* != *this* &&
> *TrSpecExpr*[[J, {*this*   *s*} ∪ wt, {}]]
> ==> *TrSpecExpr*[[J, {*this*   *s*}, {}]])

#**else**
> **postcondition** *Free* : (ALL *s* :: *TypeCorrectAs*[[*s*, U]] && *s* != *null* &&
> *TrSpecExpr*[[J, {*this*   *s*} ∪ wt, {}]]
> ==> *TrSpecExpr*[[J, {*this*   *s*}, {}]])

#**end**

where *s* is a fresh name. As an important optimization, this **postcondition** clause is suppressed for J if the free variables of J are disjoint from the domain of wt.

What should be the trigger for these quantifications?

7.4 ExtendSpecForBody

This section describes, for a given call specification spec of the form

> **spec** T m(p1 … pn) **throws** {X1 … Xx}
> **precondition** P1 … **precondition** Pj
> **targets** D1 … Dk
> **whole-targets** wt
> **postcondition** Q1 … **postcondition** Qm

and a scope Sc and a list of variables (syntactic targets) SynTargs, the various components of the result of *ExtendSpecForBody*[[spec, Sc, SynTargs]]. Function *ExtendSpecForBody* returns a method specification like spec but extended with additional **postcondition** clauses. These postconditions arise from object invariants and static invariants.

7.4.0 Adding preconditions

The specification returned by *ExtendSpecForBody* includes the following **precondition** clauses in addition to the **precondition** clauses in spec.

The preconditions generated here are used only in generating the verification for a body. In this context, the predicates of all preconditions are assumed and the error names are ignored. Since the error names are ignored, we have written them as *Free*.

For every Sc-visible static invariant J, *ExtendSpecForBody*[[spec, Sc, SynTargs]] includes

precondition *Free* : *TrSpecExpr*[[J]]

For every Sc-visible object invariant J of any class U, *ExtendSpecForBody*[[spec, Sc, SynTargs]] includes

precondition *Free* : (ALL *s* :: *TypeCorrectAs*[[*s*, U]] && *s* != *null*
==> *TrSpecExpr*[[J, {*this* *s*}, {}]])

7.4.1 Adding postconditions

The specification returned by *ExtendSpecForBody* includes the following **postcondition** clauses in addition to the **postcondition** clauses in spec.

For every Sc-visible static invariant J, *ExtendSpecForBody*[[spec, Sc, SynTargs]] includes

postcondition *StaticInvariantViolation* : *TrSpecExpr*[[J]]

As an important optimization, this **postcondition** clause is suppressed for J if the free variables of J are disjoint from SynTargs (in which case the condition follows immediately from the assumption, placed in the scope-specific background predicate, that all object invariants hold initially).

For every Sc-visible object invariant J of any class U, *ExtendSpecForCall*[[spec, Sc, SynTargs]] includes

#**if** (m is a constructor, and U is a proper subtype of T)
postcondition *ObjectInvariantViolation* :
(ALL *s* :: *TypeCorrectAs*[[*s*, U]] && *s* != *null* && *s* != *this*
==> *TrSpecExpr*[[J, {*this* *s*}, {}]])
#**else**
postcondition *ObjectInvariantViolation* :
(ALL *s* :: *TypeCorrectAs*[[*s*, U]] && *s* != *null* &&
==> *TrSpecExpr*[[J, {*this* *s*}, {}]])
#**end**

where *s* is a fresh name. As an important optimization, this **postcondition** clause is suppressed for J if the free variables of J are disjoint from SynTargs.

What should be the trigger for these quantifications?

8 Verification conditions

A verification condition consists of a set of background axioms (described in ESCJ 8, *The logic of ESC/Java*), a class-specific (that is, scope-specific) background predicate, and method-specific predicate.

8.0 Scope-specific background predicate

In this section, we define two functions, *PreMap* and *InitialState*.

Given a scope Sc, *PreMap*[[Sc]] returns a map from every field in Sc, and from *elems* and from *alloc*, to corresponding variables adorned with @pre.

As a side effect, *PreMap* creates AST nodes for these adorned variables.

PreMap[[Sc]] ==
#**for** every static field g visible in Sc **do**
{g g@pre} ∪
#**end**
#**for** every instance variable f visible in Sc **do**

{f   f@pre} ∪
#**end**
{*elems*   *elems*@pre} ∪
{*alloc*   *alloc*@pre}

The scope-specific background predicate is generated by the function *InitialState*[[Sc, premap]]. It is defined as follows, for any scope Sc and map from variables to variables premap,

InitialState[[Sc, premap]] ==
#**for** every static field g visible in Sc **do**
premap[[g]] == g &&
TypeCorrect[[g]] &&
#**end**

#**for** every instance variable f of type T visible in Sc **do**
premap[[f]] == f &&
FieldTypeCorrect[[f]] &&
#**end**

premap[[*elems*]] == *elems* &&
ElemsTypeCorrect[[*elems*]] &&

LS == *asLockSet*(*LS*) &&

premap[[*alloc*]] == *alloc*

8.1 Methods and constructors

This section describes the verification condition for a method or constructor m with a Java body S in scope Sc.

Let premap denote *PreMap*[[Sc]], let body denote *TrBody*[[m, S, premap]] (defined below), let SynTargs denote *NTargets*[[body, {}]], and let spec denote the method specification

spec T m(p1 … pn) **throws** {X1 … Xx}
precondition P1 … **precondition** Pj
targets D1 … Dk
whole-targets wt
postcondition Q1 … **postcondition** Qm

returned by *GetSpecForBody*[[m, Sc, SynTargs]]. Then, the verification condition for m declared at location L in scope Sc with body S is:

BackgroundAxioms[[Sc]] &&
InitialState[[Sc, premap]] &&
P1 && … && Pj
==>
ejp[[body ;
check L, Q1 ; … ; **check** L, Qm ;
CheckModifiesConstraints[[spec, Sc, SynTargs, premap]]
, *true*, *true*]]

Since *PreMap* has side effects (it allocates AST nodes for the variables in its image of the map it returns), the implementation must call PreMap at most once per verification condition (that is, it must use the same premap in the calls to *InitialState* and *CheckModifiesConstraints* above). The implementation will benefit from calling *PreMap* (and *InitialState*) only once per scope, that is, the results of *PreMap* and *InitialState* can be shared among the methods in one class.

We now define *TrBody* and *CheckModifiesConstraints*.

Function *TrBody* translates the Java body S into a guarded command.

```
TrBody[[ m, S, premap ]] ==
 #var V = {}, CS in
  #if (m is a method, not a constructor)
   #if (m is a synchronized instance method)
    #CS = TrSynchronizedBody[[ this, S, V ]]
   #elsif (m is a synchronized static method of a class U)
    #CS = TrSynchronizedBody[[ U, S, V ]]
   #else
    #CS = TrStmt[[ S, V ]]
   #end
   Note that constructors cannot be synchronized [JLS, 8.6.3].
  #elsif (m is a constructor of a class T, and
          S has the form construct m´ (…) ; S´ (where S´ may be the empty statement) )
   #if (m´ is a constructor of class T)
    // this is a call to a sibling constructor
    #CS = ( TrStmt[[ construct m´ (E1 … En), V ]] ;
            assume typeof(this) <: T ;
            TrStmt[[ S´, V ]] )
   #else
    // this is a call to a superclass constructor
    #CS = ( TrStmt[[ construct m´ (E1 … En), V ]] ;
            assume typeof(this) <: T ;
            InstanceInitializers[[ T, V ]] ;
            TrStmt[[ S´, V ]] )
   #end
  #else
   // this is a constructor of class Object that does not call any sibling constructor
   #CS = ( modify this ; modify alloc ;
           assume premap[[ alloc ]] < alloc ;
           assume premap[[ alloc ]] < allocTime(this) && allocTime(this) < alloc ;
           assume this != null && typeof(this) <: Object ;
           InstanceInitializers[[ Object, V ]] ;
           TrStmt[[ S, V ]] )
  #end

  var p1@pre … pn@pre in
   Note, the parameters p1 … pn are not in the domain of premap.  The AST nodes for these @pre variables are thus allocated
   here.
   p1@pre = p1 ; … ; pn@pre = pn ;
   var V in
    ( CS ; EC = ec$return
     #if (m is a constructor)
      ; RES = this
     #end
    ) ! skip
   end ;
   p1 = p1@pre ; … ; pn = pn@pre
  end
 #end
```

where for any class type T,

```
InstanceInitializers[[ T, V ]] ==
 #for every instance variable f of type U declared in class T in order do
  #if (T is boolean)
   assume f[this] != bool$true ;
```

```
  #elsif (T is an integral type)
    assume f[this] == 0 ;
  #elsif (T is a reference type)
    assume f[this] == null ;
  #elsif (T is a floating point type)
    assume f[this] == cast(0, T) ;
  #end
#end
#for every instance variable f with an initializer E declared in class T in order do
  #var e in
    TrExpr[[ E, {}, V, e ]] ;
    WriteCheck[[ f[this], e ]] ;
    f[this] = e
  #end
#end
```

Function *CheckModifiesConstraints* takes a list of designator targets, a whole-targets map, and a list of syntactic targets, and produces a sequence of **check** commands. These checks enforce that the body meets the **modifies** list of the specification. Let spec be a method specification of the form shown above. Then, for a scope Sc and a list of variables (syntactic targets) SynTargs:

```
CheckModifiesConstraints[[ spec, Sc, SynTargs, premap ]] ==
 #for every static field g in SynTargs and not in Domain[[ wt ]] do
       check ModifiesViolation :  premap[[ g ]] == g
 #end
 #for every instance variable f in SynTargs do
   #let U be the class that declares f in
     #let q1 … qs be the subset of p1 … pn whose types are subtypes of U in
       #let g1 … gr be the static fields in Sc whose types are are subtypes U in
         check ModifiesViolation :
               (ALL s :: s != null && (s == q1 || … || s == qs || s == g1 || … || s == gr)
                      ==>
                        premap[[ f ]] [s] == f[s] || IsModPoint[[s, f, D1 … Dk]])
       #end
     #end
   #end
 #end
 #if (elems is in SynTargs)
   #let q1 … qs be the subset of p1 … pn whose types are array types in
     #let g1 … gr be the static fields in Sc whose types are array types in
         check ModifiesViolation :
               (ALL a :: a != null && (a == q1 || … || a == qs || a == g1 || … || a == gr)
                      ==>
                      (ALL i :: premap[[ elems ]] [a][i] == elems[a][i]) ||
                      IsArrayModPoint[[ a, D1 … Dk ]] )
         check ModifiesViolation :
               (ALL a, i :: a != null && (a == q1 || … || a == qs || a == g1 || … || a == gr)
                        ==>
                        premap[[ elems ]] [a][i] == elems[a][i] || IsIndexModPoint[[ a, i, D1 … Dk ]] )
```

Perhaps we also want to require modifications of p.arr[i], where p is a parameter, arr is a field of p, and i is some index into p.arr, to be explicitly mentioned in a **modifies** clause. If so, we should add some more disjuncts of the form *a* == p.arr.

```
     #end
   #end
 #end
```

We now define *IsModPoint*, *IsArrayModPoint*, and *IsIndexModPoint*.

For any name s, instance variable name f, and specification designator list D1 … Dk, function *IsModPoint* produces a predicate stating that the **modifies** list D1 … Dk allows f to be modified at s:

IsModPoint[[s, f, D1 … Dk]] ==
  #**if** (k == 0)
    false
  #**elsif** (D1 has the form f[E] for some E)
    s == E || *IsModPoint*[[s, f, D2 … Dk]]
  #**else**
    IsModPoint[[s, f, D2 … Dk]]
  #**end**

For any names a and i, and specification designator list D1 … Dk, function *IsArrayModPoint* produces a predicate stating that the **modifies** list D1 … Dk allows *elems* to be modified at a, and function *IsIndexModPoint* produces a predicate stating that the **modifies** list D1 … Dk allows *elems*[a] to be modified at i:

IsArrayModPoint[[a, D1 … Dk]] ==
  #**if** (k == 0)
    false
  #**elsif** (D1 has the form *elems*[E0][E1] for some E0 and E1)
    (a == E0) || *IsArrayModPoint*[[a, D2 … Dk]]
  #**elsif** (D1 has the form *elems*[E] for some E)
    (a == E) || *IsArrayModPoint*[[a, D2 … Dk]]
  #**else**
    IsArrayModPoint[[a, D2 … Dk]]
  #**end**

IsIndexModPoint[[a, i, D1 … Dk]] ==
  #**if** (k == 0)
    false
  #**elsif** (D1 has the form *elems*[E0][E1] for some E0 and E1)
    (a == E0 && i == E1) || *IsIndexModPoint*[[a, i, D2 … Dk]]
  #**elsif** (D1 has the form *elems*[E] for some E)
    (a == E) || *IsIndexModPoint*[[a, i, D2 … Dk]]
  #**else**
    IsIndexModPoint[[a, i, D2 … Dk]]
  #**end**

8.2 Static bodies

TBW.