

Joining Forces! Reusing Contracts for Deductive Verifiers through Automatic Translation[★]

Lukas Armbrorst^[0000–0001–7565–0954], Sophie Lathouwers^[0000–0002–7544–447X],
and Marieke Huisman^[0000–0003–4467–072X]

University of Twente, Enschede, the Netherlands
`{l.armbrorst,s.a.m.lathouwers,m.huisman}@utwente.nl`

This is supplemental material to the iFM 2023¹ paper of the same title. For the paper itself, please refer to the conference proceedings.

[★] This work was supported by the NWO VICI 639.023.710 Mercedes project.

¹ <https://liacs.leidenuniv.nl/~bonsanguemm/ifm23/index.html>

Appendix: Grammar of Intermediate Representation (IR)

This appendix describes the Intermediate Representation (IR) used by the SPECIFICATION TRANSLATOR. The IR is described via a grammar, using BNF and regular expressions. In particular, terminal symbols are shown like `this`, while non-terminal symbols are shown as `ruleName`. We use the star to indicate 0 or more repetitions (like “`ruleName*`”), the plus to indicate 1 or more repetitions (like “`ruleName+`”), and square brackets to indicate optional parts (like “`ruleName [‘,’ ruleName]`”). Choices are indicated with a vertical bar `|`, and parentheses group elements, especially in combination with the repetition star `*`. Comments about the grammar, for instance notes about deviations from the JML grammar, are shown as `// notes like this`.

The IR is used as a syntax tree within the Specification Translator, and this grammar is only for illustration purposes. In particular, there is no parser associated with this grammar, to transform text directly into the IR. As a result, things like operator precedence are not part of the grammar. Instead, parsers for input languages have to handle this, and encode the precedence via the node hierarchy in the syntax tree. Additionally, the implementation, and thus the grammar, at times over-approximate. For instance, the implementation might store a list of nodes, which means 0 or more elements, while in reality only non-empty lists are reasonable and allowed by frontends.

Every node in the syntax tree has two fields, `precedingWhitespace` and `trailingWhitespace`, that are of type *String* and that can contain comments, whitespace and other parts that are not relevant for translating specifications. These are just copied over during the translation. Consequently, these parts are not mentioned explicitly in the grammar, but are implicitly included before and after any terminal or non-terminal. It is the responsibility of the input language parser to ensure that these fields only contain information that is truly irrelevant for the translation, for example Java line comments starting with a double slash `//` and ending with a newline character. The IR does not impose any format or other restrictions on these strings; in practice, the fields can contain any string.

Java Nodes

```

compilationUnit ::= [ packageDeclaration ] importDeclaration* typeDeclaration*
                  | nontranslatedCompilationUnit
nontranslatedCompilationUnit ::= string
packageDeclaration ::= annotation* 'package' string ';'
importDeclaration ::= 'import' modifier* string ';'
                  | startSpec ( 'model' 'import' modifier* string ';' )+ endSpec
typeDeclaration ::= classDeclaration
                  | interfaceDeclaration
                  | enumDeclaration
                  | nontranslatedTypeDeclaration
                  | specDeclarationBlock

```



```

    | annotationTypeDeclaration
    | ';'
nontranslatedTypeDeclaration ::= string
classDeclaration ::= [ contract ] modifier* 'class' string
    [ typeParameterClause ] [ superclassClause ] [ interfaceClause ]
    body
interfaceDeclaration ::= [ contract ] modifier* 'interface' string
    [ typeParameterClause ] [ multiSuperclassClause ]
    body
enumDeclaration ::= [ contract ] modifier* 'enum' string
    [ interfaceClause ]
    '{' [ enumConstant ( ',' enumConstant )* ] [ ',' ] [ ';' ] memberDeclaration*
    '}'
enumConstant ::= annotation* string [ arguments ] [ body ]
superclassClause ::= 'extends' typeIndication
interfaceClause ::= 'implements' [ typeIndication ( ',' typeIndication )* ]
multiSuperclassClause ::= 'extends' [ typeIndication ( ',' typeIndication )* ]
typeParameterClause ::= '<' [ typeParameter ( ',' typeParameter )* ] '>'
typeParameter ::= annotation* typeIndication [ genericsSuperclassClause ]
genericsSuperclassClause ::= 'extends' [ typeIndication ( '&' typeIndication )* ]
body ::= '{' memberDeclaration* '}'
modifier ::= 'static' | string | annotation | jmlModifier
annotation ::= stringAnnotation // string is currently the only supported type,
    may change in future
stringAnnotation ::= string
annotationTypeDeclaration ::= '@' 'interface' string body
memberDeclaration ::= methodDeclaration
    | classDeclaration
    | interfaceDeclaration
    | enumDeclaration
    | fieldDeclaration
    | annotationTypeDeclaration
    | blockOrSemicolon
    | nontranslatedMemberDeclaration
    | specDeclaration
nontranslatedMemberDeclaration ::= string
methodDeclaration ::= [ contract ] modifier* [ typeIndication ] string
    '(' [ parameter ( ',' parameter )* ] ')'
    [ string ] // this string indicates array dimensions via multiple "[]"
    [ throwsClause ] [ defaultClause ] [ contract ]
    blockOrSemicolon
blockOrSemicolon ::= block | ';'
throwsClause ::= 'throws' [ string ( ',' string )* ]
defaultClause ::= 'default' string // currently only string, as it is only for
    annotations

```



```

parameter ::= modifier* typeIndication string
           | modifier* typeIndication annotation* '...' string
typeArgumentClause ::= '<' [ typeArgument ( ',' typeArgument )* ] '>'
typeArgument ::= annotation* '?' [ 'super' typeIndication ] [ 'extends' typeIndication
           ]
           | typeIndication
typeIndication ::= annotation* modifier* string
           | "// can explicitly be the empty string, e.g. for lambda parameters without type"
fieldDeclaration ::= modifier* typeIndication
           [ variableDeclarator ( ',' variableDeclarator )* ] ';'
variableDeclarator ::= string [ '=' initializer ]
initializer ::= expression | arrayInitializer
arrayInitializer ::= '{' [ initializer ( ',' initializer [ ',' ] )* ] '}'
statement ::= block
           | localDeclaration
           | javaAssertion
           | ifStatement
           | loop
           | tryBlock
           | switchStatement
           | synchronizedStatement
           | returnStatement
           | throwStatement
           | breakStatement
           | continueStatement
           | ';'
           | expression ';'
           | labeledStatement
           | jmlStatement
           | typeDeclaration
           | nontranslatedStatement
nontranslatedStatement ::= string
block ::= [ modifier ] '{' statement* '}' // modifier intended for static
localDeclaration ::= modifier* typeIndication [ variableDeclarator ( ',' variableDeclarator
           )* [ ';' ] // semicolon is present if this is a statement
javaAssertion ::= 'assert' expression [ ':' expression ] ';'
ifStatement ::= 'if' '(' expression ')' statement [ 'else' statement ]
loop ::= forLoop | whileLoop | doLoop
forLoop ::= loopSpecOrLabel* 'for' '(' forLoopControl ')' loopSpecOrLabel*
           body
forLoopControl ::= regularForLoopControl | forEachLoopControl
regularForLoopControl ::= [ forInit ] ';' [ expression ] ';' [ expression ]
forInit ::= localDeclaration | [ expression ( ',' expression )* ]
forEachLoopControl ::= modifier* typeIndication string ':' expression

```



```

whileLoop ::= loopSpecOrLabel* 'while' '(' expression ')' loopSpecOrLabel*
            body
doLoop ::= loopSpecOrLabel* 'do' body 'while' '(' expression ')' ';'
tryBlock ::= 'try' [ tryResourceSpec ] block catchClause* [ 'finally' block ]
catchClause ::= 'catch' '(' modifier* [ typeIndication ( '|' typeIndication )* string
                ')' block
tryResourceSpec ::= '(' [ tryResource ( ';' tryResource )* ] [ ';' ] ')'
tryResource ::= modifier* typeIndication string '=' expression
switchStatement ::= 'switch' '(' expression ')' '{' switchCaseBlock* '}'
switchCaseBlock ::= switchLabel* statement*
switchLabel ::= 'case' expression ':' | 'default' ':'
synchronizedStatement ::= 'synchronized' '(' expression ')' statement
returnStatement ::= 'return' [ expression ] ';'
throwStatement ::= 'throw' expression ';'
breakStatement ::= 'break' [ string ] ';'
continueStatement ::= 'continue' [ string ] ';'
labeledStatement ::= label statement
label ::= string ':'
arguments ::= '(' [ expression ( ',' expression )* ] ')'

```

Expression Nodes

```

expression ::= nestedInvocationExpression
            | binaryExpression
            | instanceOfExpression
            | arrayIndexExpression
            | methodCallExpression
            | newObjectExpression
            | newArrayExpression
            | castExpression
            | postfixExpression
            | prefixExpression
            | ternaryExpression
            | lambdaExpression
            | methodReferenceExpression
            | parExpression
            | literal
            | className
            | genericInvocationExpression
            | jmlExpression
nestedInvocationExpression ::= expression '.' expression
binaryExpression ::= expression string expression
instanceOfExpression ::= expression 'instanceof' typeIndication
arrayIndexExpression ::= expression '[' indexExpression ']'
indexExpression ::= expression | jmlIndexExpression

```



```

methodCallExpression ::= string arguments
newObjectExpression ::= 'new' [ typeArgumentClause ] typeIndication arguments
    [ body ]
newArrayExpression ::= 'new' typeIndication dimension* [ arrayInitializer ]
dimension ::= '[' expression ']'
castExpression ::= '(' annotation* [ typeIndication ( '&' typeIndication )* ] ')'
    expression
postfixExpression ::= expression string
prefixExpression ::= string expression
ternaryExpression ::= expression '?' expression ':' expression
lambdaExpression ::= '(' [ parameter ( ',' parameter )* ] ')' '->' lambdaBody
    [ [ parameter ( ',' parameter )* ] '->' lambdaBody
lambdaBody ::= expression | block
methodReferenceExpression ::= expression '::' [ typeArgumentClause ] expression
parExpression ::= '(' expression ')'
literal ::= string
className ::= typeIndication '.' 'class'
genericInvocationExpression ::= typeArgumentClause string [ arguments ]
jmlIndexExpression ::= '*' | rangeExpression
rangeExpression ::= expression '..' expression

```

Specification Nodes

```

jmlModifierBlock ::= [ startSpec ] modifier* [ endSpec ]
jmlModifier ::= 'non_null' | 'nullable' | 'nullable_by_default'
    | 'model' | 'ghost' | 'pure'
    | 'instance' | 'helper'
    | 'monitored' | jmlModifierBlock | string
contract ::= [ startSpec ] contractClause* [ endSpec ]
contractClause ::= [ startSpec ]
    ( specCase
        | assignableClause | accessibleClause
        | forallVariableDeclaration | oldVariableDeclaration
        | precondition | postcondition
        | exceptionalPostcondition | signalsOnlyClause
        | measuredByClause | divergeClause
        | behavior
    ) [ endSpec ]
    | nontranslatedContractClause
nontranslatedContractClause ::= string
specCase ::= [ 'also' ] contractClause*
assignableClause ::= 'assignable' [ expression ( ',' expression )* ] ';'
accessibleClause ::= 'accessible' [ expression ( ',' expression )* ] ';'
forallVariableDeclaration ::= 'forall' quantifiedVarDeclaration ';'
oldVariableDeclaration ::= 'old' modifier* typeIndication [ variableDeclarator (
    ',' variableDeclarator )* ] ';'

```



```

precondition ::= 'requires' expression ';'
postcondition ::= 'ensures' expression ';'
exceptionalPostcondition ::= 'signals' '(' typeIndication [ string ] ')' [ expression
] ';'
signalsOnlyClause ::= 'signals_only' [ typeIndication ( ',' typeIndication )* ]
                        ';'
                        | 'signals_only' '\nothing' ';'
divergeClause ::= 'diverges' expression ';'
measuredByClause ::= 'measured_by' expression [ 'if' expression ] ';'
behavior ::= [ 'also' ] [ privacy ] ( 'behavior' | 'normal_behavior' | 'exceptional_behavior' )
contractClause*
privacy ::= string // intended for public and similar visibility modifiers
loopSpecOrLabel ::= loopSpec | label
loopSpec ::= [ startSpec ] loopClause* [ endSpec ]
loopClause ::= loopInvariant | decreasesClause | loopModifiesClause | nontranslatedLoopClause
nontranslatedLoopClause ::= string
loopInvariant ::= 'loop_invariant' expression ';'
decreasesClause ::= 'decreases' expression ';'
loopModifiesClause ::= 'loop_modifies' [ expression ( ',' expression )* ] ';'
jmlStatement ::= assertStatement | assumeStatement | ghostStatement | setStatement
               | unreachableStatement | jmlStatementBlock | foldStatement | unfoldStatement
assertStatement ::= 'assert' expression [ ':' expression ] ';'
assumeStatement ::= 'assume' expression [ ':' expression ] ';'
ghostStatement ::= 'ghost' statement
setStatement ::= 'set' expression ';'
unreachableStatement ::= 'unreachable' ';'
jmlStatementBlock ::= [ startSpec ] statement* [ endSpec ]
foldStatement ::= 'fold' expression ';'
unfoldStatement ::= 'unfold' expression ';'
jmlExpression ::= resultExpression
                | oldExpression | preExpression
                | notAssignedExpression | notModifiedExpression
                | onlyAccessedExpression | onlyAssignedExpression
                | freshExpression | nonnullelementsExpression
                | typeofExpression | typeExpression
                | quantifiedExpression | newSetComprehension
                | permissionExpression | valuedPermissionExpression
                | currentPermissionExpression | somePermissionExpression | scaledPredicateExpression
                | unfoldingExpression
                | expression '**' expression | expression '==>' expression
resultExpression ::= '\result'
oldExpression ::= '\old' '(' expression [ ',' string ] ')'
preExpression ::= '\pre' '(' expression ')'
notAssignedExpression ::= '\not_assigned' '(' expressionList ')'
notModifiedExpression ::= '\not_modified' '(' expressionList ')'
onlyAccessedExpression ::= '\only_accessed' '(' expressionList ')'

```



```

onlyAssignedExpression ::= '\only_assigned' '(' expressionList ')'
freshExpression ::= '\fresh' '(' expressionList ')'
nonnullelementsExpression ::= '\nonnullelements' '(' expression ')'
typeofExpression ::= '\typeof' '(' expression ')'
typeExpression ::= '\type' '(' typeIndication ')'
quantifiedExpression ::= '(' quantifier quantifiedVarDeclaration* ';' [ [ expression
    ] ';' ] expression ')'
quantifier ::= '\forall' | '\exists' | '\max' | '\min' | '\num_of' | '\product'
    | '\sum'
newSetComprehension ::= 'new' typeIndication '{' quantifiedVarDeclaration '|'
    expression '&&' expression '}'
permissionExpression ::= 'IR.permission_expression' '(' expression ',' [ expression
    ] ')'
valuedPermissionExpression ::= 'IR.valued_permission_expression' '(' expression
    ',' expression ',' expression ')'
currentPermissionExpression ::= 'IR.current_permission' '(' expression ')'
somePermissionExpression ::= 'IR.some_permission' '(' expression ')'
scaledPredicateExpression ::= '[' expression ']' expression
unfoldingExpression ::= '\unfolding' expression '\in' expression
expressionList ::= [ expression ( ',' expression )* ]
quantifiedVarDeclaration ::= [ ('nullable' | 'non_null' ) ] typeIndication [ string
    ( ',' string )* ]
specDeclarationBlock ::= [ startSpec ] specDeclaration* [ endSpec ]
specDeclaration ::= ghostDeclaration | modelDeclaration | invariantDeclaration |
    initiallyDeclaration | predicateDeclaration | readableIfClause | writableIfClause
    | monitorsForClause | axiomDeclaration | nontranslatedSpecDeclaration
nontranslatedSpecDeclaration ::= string
ghostDeclaration ::= 'ghost' memberDeclaration
modelDeclaration ::= 'model' memberDeclaration
invariantDeclaration ::= modifier* 'invariant' expression ';'
initiallyDeclaration ::= modifier* 'initially' expression ';'
predicateDeclaration ::= modifier* typeIndication string
    '(' [ parameter ( ',' parameter )* ] ')' [ '=' expression ] ';'
readableIfClause ::= modifier* 'readable' string 'if' expression ';'
writableIfClause ::= modifier* 'writable' string 'if' expression ';'
monitorsForClause ::= modifier* 'monitors_for' string '=' [ expression ( ','
    expression )* ] ';'
axiomDeclaration ::= 'axiom' expression ';'
startSpec ::= string // intended for /*@ or //@, but technically any string is
    possible
endSpec ::= string // intended for @*/, newline, etc., but technically any string
    is possible

```