Original version by Joseph Kiniry. Initially written in early 2004, then lost, then rewritten beginning in November, 2004.

Current author and editor: Joseph Kiniry.

This document describes how to extend ESC/Java2. It describes the high-level architecture of the system and how to extend it through a series of case studies.

This is edition $Revision: 71985 $.

This document is a work in progress. Suggestions and input are always welcome.

Extending ESC/Java2

Edition $Revision: 71985 $, November 2004
This document describes how to extend ESC/Java2 version 2.0a8 and later.

Joseph R. Kiniry <joseph.kiniry@ucd.ie>

Table of Contents

This document describes how to extend ESC/Java2. It describes the high-level architecture of the system and how to extend it through a series of case studies.

1 Introduction

This document describes how to extend ESC/Java2. It describes the high-level architecture of the system and how to extend it through a series of case studies.

The case studies discussed were chosen because they: (a) focused on each independent aspects of the architecture, (b) were relatively small and self-contained examples, and (c) they were additions that we made to the original SRC ESC/Java system, thus writing them up in this manner was straightforward.

We describe the ESC/Java2 architecture using the specification language BON. BON is a specification language developed by Kim Walden and Jean-Marc Nerson and is documented in the book "Seamless Object-Oriented Software Construction," available from http://www.bon-method.com/.

BON is a simple enough specification language that most, if not all, of the specifications herein will be understandable to the reader who knows nothing at all of BON.

2 The ESC/Java2 Architecture

ESC/Java2 is an extension of Digital/Compaq/Hewlett-Packard System Research Center's (SRC henceforth) ESC/Java.

SYSTEM	EXTENDED_STATIC_CHECKER	Part: 1/1

PURPOSE
 An extended static checker for Java.

INDEXING
 author: Joseph Kiniry
 keywords: extended static checker, ESC, Java

Cluster	Description
JAVA_FRONT_END	A generic Java scanner, parser, and type checker.
EXTENDED_STATIC_CHECKER	An extended static checker for Java.
UTILITY	Miscellaneous utility classes.

Chart 2.1: The ESC/Java2 System

ESC/Java2 has two primary subcomponents: a generic Java front-end in the Java package *javafe* and the extended static checking framework found in the package *escjava*.

There are a few other classes used for testing found in the *junitutils* package. We discuss this part of the architecture in Section 8.9 [jUnit Tests], page 18.

ESC/Java depended at one time (and perhaps still today) upon a few external packages that we do not have the source for. These relevant packages are called *mochalib*, *decsrc*, *jtools*, and *tohtml*[1]

ESC/Java2 depends upon a number of Open Source Java libraries whose Java archive (jar) files are provided with the source release. The current packages on which we depend include Ant, jUnit, and XML-RPC libraries.

2.1 The Generic Java Front-end

ESC/Java2 uses the generic Java front-end developed by SRC, available as an independent download from http://todo. The only modifications that have been made to the javafe package are the addition of support for the assert statement that was added to the Java languange in the Java 1.4 release. We discuss this extension as the first case study, found in Section 3.5.1 [The Assert Statement], page 9.

CLUSTER	JAVA_FRONT_END	Part: 1/1

PURPOSE
 A generic Java scanner, parser, and type checker.

INDEXING
 author: Joseph Kiniry
 keywords: Java, scanner, parser, type checker

Class/(Cluster)	Description
(AST)	The abstract symbol tree classes for Java.
(FILES)	TBD
(LEXER)	TBD
(OPTIONS)	TBD
(PARSER)	TBD
(READER)	TBD
(TYPECHECKER)	TBD
(UTILITY)	TBD
(TOOLS)	TBD
(UTILITY)	TBD

Chart 2.2: The ESC/Java2 Java Front-end

---

[1] There are also a number of deprecated packages currently included in the CVS repository for ESC/Java2 including *escwizard*, *houdini*, and *instrumenter*. These packages are not used or supported at this time and we plan on removing them from the CVS HEAD in the near future. See Bug #X for more information.

The Java front-end is summarised in the informal cluster chart in Chart 2.3. The main subcomponents of the Java front-end are the core tool classes (in the TOOLS cluster), a Java lexer, a parser, and type checker (the LEXER, PARSER, and TYPECHECKER clusters). An abstract symbol tree is used to represent the parsed Java classes (the AST cluster). Command-line option parsing is another major subcomponent that most extensions need to modify (the OPTIONS cluster). Finally, there are a number of classes used to search for source and class files in the classpath (FILES), read in source and class files and cache their locations and contents (READER), and track warning and errors and related data that are detected during compilation (UTILITY).

2.2 The Extended Static Checker for Java

The extended static checker for Java has several major subcomponents.

CLUSTER	EXTENDED_STATIC_CHECKER	**Part:** 1/1
PURPOSE		**INDEXING**
An extended static checker for Java.		**author:** Joseph Kiniry
		keywords: extended static checker, Java

Class/(Cluster)	**Description**
(ANT)	TBD
(AST)	TBD
(BACKGROUND_PREDICATE)	TBD
(GUI)	TBD
(OPTIONS)	TBD
(PREDICATE_ABSTRACTION)	TBD
(PARSER)	TBD
(PROVER)	TBD
(READER)	TBD
(SOUNDNESS_CHECKER)	TBD
(DYNAMIC_SINGLE_ASSIGNMENT)	TBD
(TYPECHECKER)	TBD
(VERIFICATION_CONDITION_GENERATOR)	TBD
(TOOLS)	TBD

Chart 2.3: The ESC/Java2 Extended Static Checker

A full JML parser and type checker is the first main component (clusters PARSER and TYPE-CHECKER). It is realised as a specialisation of the generic pragma parser in the Java front-end (cluster JAVA_FRONT_END.PARSER). Conseqently, an AST hierarchy also exists for JML constructs (cluster AST).

3 The Java Front-end Architecture

```
static_diagram Tag_Constants
```
3.1 The Java Lexer
```
static_diagram Tag_Constants
-- The inheritance relationship between all of the classes that define
-- lexical tags.
component
  -- Classes with a GENERATED_ prefix are generated automatically from
  -- the a hierarchy.j class using the astgen tool.  See the section of
  -- this manual on astgen (node "The Java AST") for more information.
  class JAVAFE.AST.GENERATED_TAGS
  class JAVAFE.AST.OPERATOR_CONSTANTS
  class JAVAFE.AST.TAG_CONSTANTS

  class JAVAFE.PARSER.TAG_CONSTANTS

  class JAVAFE.TYPECHECKER.TAG_CONSTANTS

  class ESCJAVA.AST.TAG_CONSTANTS
  class ESCJAVA.AST.GENERATED_CONSTANTS

  JAVAFE.AST.OPERATOR_CONSTANTS inherit JAVAFE.AST.GENERATED_TAGS
  JAVAFE.AST.TAG_CONSTANTS inherit JAVAFE.AST.OPERATOR_CONSTANTS

  JAVAFE.PARSER.TAG_CONSTANTS inherit JAVAFE.AST.TAG_CONSTANTS
  JAVAFE.TYPECHECKER.TAG_CONSTANTS inherit JAVAFE_PARSER.TAG_CONSTANTS

  ESCJAVA.AST.GENERATED_CONSTANTS inherit JAVAFE.TYPECHECKER.TAG_CONSTANTS
  ESCJAVA.AST.TAG_CONSTANTS inherit ESCJAVA.AST.GENERATED_CONSTANTS
end
```
Diagram 3.1: Constants Representing Fundamental Lexing and Parsing Constructs

The Java lexer that is part of ESC/Java2 is, like the Java and pragma parsers, hand-written. This means that it is faster and smaller than a generated lexer, but is also a bit trickier to understand and extend.

The Java lexer inherits from a token class that describes the lexical tokens of the parser. The lexer provides arbitrary lookahead using a token queue. The static diagram in Diagram 3.2 summarises the classes related to the Java lexer.

```
static_diagram Java_Lexer
component
  cluster JAVAFE.PARSER
  component
    class TAG_CONSTANTS

    class FILE_FORMAT_EXCEPTION

    class LEXER
    class TOKEN
    class TOKEN_QUEUE

    TAG_CONSTANTS inherit JAVAFE.AST.TAG_CONSTANTS

    FILE_FORMAT_EXCEPTION inherit JAVA.IO.IO_EXCEPTION

    LEXER inherit TOKEN
  end
end
```

Diagram 3.2: The Java Lexer

A complementary hierarchy of tag constant classes define all of the lexical constants of the lexer. The static diagram in Diagram 3.1.

CLASS	TOKEN		**Part:** 1/1

PURPOSE	**INDEXING**
An abstract representation of a Java token.	**author:** Joseph Kiniry
	cluster: JAVAFE.PARSER
	keywords: token, lexer, Java

Queries	"What is the current token?", "What is the location of the first character of the current token?", "What is the location of the last character of the current token?", "What is the identifier related to the current token, if it is an identifer token?", "What auxilliary information is available for the current token?", "What is the string representation of the current token?", "What is the full string representation of the current token, suitable for debugging output?", "Does the current token fulfill its invariants?"
Commands	"Clear the current token.", "Copy all the fields from a source token into a target token."
Constraints	"The start and ending location of a token must be valid.", "If the current token represents a Java integral type literal, then the auxilliary information must only be value or the corresponding integral type.", "If the current token represents a Java character or string type, then the auxilliary information must only be a value or the corresponding Java value type.", "If the current token represents a lexical, modifier, statement, or type declaration element pragma, then the auxilliary information must only be a value of the corresponding pragma type."

Chart 3.1: The Java TOKEN Class Chart

CLASS	LEXER		**Part:** 1/1

PURPOSE	**INDEXING**
Generates a sequence of Java tokens by converting a sequence of input characters and line terminators.	**author:** Joseph Kiniry
	cluster: JAVAFE.PARSER
	keywords: lexer, Java

Queries
Commands
Constraints

Chart 3.2: The Java LEXER Class Chart

3.2 The Java AST

3.2.1 AST Generation

A custom tool called `astgen` is used to generate the Java and JML abstract symbol tree classes. `astgen` reads an input file that is a kind of "Java shorthand" with annotations and it generates JML-annotated Java source code.

The input to `astgen` must have the same lexical language as Java and must follow the following grammar: `astfile ::= PackageDeclaration_opt ImportDeclarations_opt ClassDeclaration*` where the non-terminals on the right are those defined in the Java Language Specification. Before the first `ClassDeclaration` there must be a line with the following lexical structure `^[white-space]*"//#"[white-space]*"EndHeader".*\n`

Roughly speaking, `astgen` does the following: all text (including comments and whitespace) before the `EndHeader` directive is read into a buffer. Then, for each `ClassDeclaration`, the text of the declaration (again including comments and whitespace) is appended to an "expanded" version of each generated class decaration.

AST generation is stateful. The module responsible for keeping track of the state of `astgen` is called `astactions()` (it implements a state machine). The state transitions of this machine are:

```
<anystate> -(init)-> INIT
INIT -(visitorroot)-> INIT
INIT -(tagbase)-> INIT
INIT -(endheader)-> ABOVECLASS
INIT -(endastfile)-> DONE
ABOVECLASS -(abstract)-> ABOVECLASS
ABOVECLASS -(endastfile)-> DONE
ABOVECLASS -(classname)-> SUPERLESS -(supername)-> SUPERFULL
SUPERCLESS,SUPERFULL -(beginclass)-> INCLASS
INCLASS -(endclass)-> ABOVECLASS
```

The state machine starts in the special state `UNINITIALIZED`. The `init()` function is the only routine that may be called when the module is in this state.

With two exceptions, every piece of text in the input file (except the EOF character) is sent through (exactly) one of the following echo routines. This includes text that is also passed to state transition procedures such as "`classname()`" and "`supername()`"; in these cases, the function `astecho()` is called first, then the state transition routine.

One exception to the rule that every piece of text is sent to an echo routine is the line containing the "`//# EndHeader`" section, which triggers the call to `endheader()` function. The other exception is the '`}`' character that ends a (top-level) class declaration; this triggers the call to "`endclass()`" function, to which the closing '`}`' plus any characters matching "`"[white-space]\n{0,1}`" is passed.

The `astecho()` function is called in most situations; it may be called in any state. The `expand()` function is called only in state `INCLASS`, and is called only on (and on every) piece of text that matches the pattern `[whitespace]*"//#".*\n`.

`astgen` is written in C. A scanner/lexer is generated using `lex`. The lexer is quite simple, as it only matches the basic structure of Java with special annotations prefixed by `//#`. The source for `astgen` is located in the directory '`ESCTools/Javafe/astgen`'.

There are six different kinds of annotations, also known as "directives" in the `astgen` source code, organised into three different categories.

The first type of directive are field qualifiers, as they are used to annotate Java fields. These annotations include:

- `NullOK` indicates that the annotated field, which must be a reference type, may have a `null` value. Such fields are annotated in the generated source with the `nullable` JML annotation. (Prior to the introduction of this annotation, all fields *not* annotated with `NullOK` were labeled with the `non_null` JML annotation.)

- `NoCheck` is used to indicate that a field's invariant should *not* be checked when its enclosing class's invariant is checked.

- `NotNullLoc`

- `Syntax`

The character "`*`" is also used as a special annotation on types to indicate the multiplicity of a field.

- `NoMaker`

- `ManualTag`

- `PostMakeCall`
- `PostCheckCall`
- `MakerSpec`

Finally, the annotation `EndHeader` is used to indicate the end of the AST class declaration.

The source file 'astutil.h' summarises these annotations.

The balance of the Java code, fields that are not annotated with `astgen` comments, can be annotated with JML specifications and ESC/Java2 pragmas and `astgen` will ignore them completely.

3.3 The Java Parser

```
static_diagram Java_Parser
component
  cluster JAVAFE.PARSER
  component
    class PARSE
    class PARSE_EXPRESSION
    class PARSE_STATEMENT
    class PARSE_TYPE
    class PARSE_UTILITY

    deferred class PRAGMA_PARSER

    PARSE inherit PARSE_STATEMENT
    PARSE_STATEMENT inherit PARSE_EXPRESSION
    PARSE_EXPRESSION inherit PARSE_TYPE
    PARSE_TYPE inherit PARSE_UTILITY
  end
end
```
Diagram 3.3: Java Parser Classes and Inheritance

The Java parser is decomposed into several classes related to each other via inheritance, summarised in the static diagram in Diagram 3.3. A pragma parser interface is also defined from which one must inherit to parse pragmas from pragma-containing comments. It is also used to check to see whether or not a comment contains pragmas in the first place.

3.4 The Java Type Checker

3.5 Case Studies in Extending the Java Front-end

3.5.1 The Assert Statement

3.5.2 New Java Bytecodes

4 The Extended Static Checker Architecture

Like in the Java front-end, the AST for pragmas is specified with a 'hierarchy.j' file which is transformed with the `astgen` tool into source Java files. This file and its related classes are locate in the `escjava.ast` package that is located in the repository at 'ESCTools/Escjava/java/escjava/ast/'.

There are a few classes in this package that are not specified using 'hierarchy.j': 'DerivedMethodDecl.java', 'EscPrettyPrint.java', 'TagConstants.java', and 'Utils.java'.

The case study that we use to demonstrate how to extend the JML pragma lexer and parser is the addition of the `nullable` keyword, presented by Patrice Chalin at SAVCBS 2005*P. Chalin and F. Rioux, "Non-null References by Default in the Java Modeling Language." Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05), Lisbon, Portugal, Sept. 2005.* (Updated version: http://www.cs.concordia.ca/%7Echalin/papers/TR-2005-004-r3.2.pdf ENCS-CSE TR 2005-004, December 2005). We discuss this extension as the in Section 4.4.1 [Default Reference Type Values], page 10.

4.1 The ESC/Java2 JML Lexer

4.2 The ESC/Java2 JML Parser

4.3 The ESC/Java2 Type Checker

4.4 Case Studies in Extending the Extended Static Checker Lexer, Parser, and Type Checker

4.4.1 Default Reference Type Values

5 The ESC/Java2 VC Generation Calculi

5.1 The DSA Intermediate Representation

5.2 The Weakest-Precondition Calculus

5.3 The Strongest-Postcondition Calculus

5.4 Case Studies in Extending the VC Generation Calculi

5.4.1 Reasoning about Loops

5.4.2 Reasoning about Java Bytecodes

6 The ESC/Java2 Logics

7 The Theorem Prover Interface

8 Testing Extensions

Whenever a new extension to ESC/Java2 is written new tests must be written to both test the new features and to ensure that existing features are not broken. Thus, tests included are used as unit tests and for regression testing.

Tests are organised into several different directories based upon their purpose. For the Java front-end there are test suites for the Java lexer, parser, and typechecker as well as for the full front-end. The JML parser and typechecker in the ESC/Java2 package also have test suites. The extended static checker has several test suites including a core set of reasoning tests, a jUnit test suite that focuses on different facets of verification, and a test suite that focuses on extended static checking with rich JML specifications of core Java Developers Kit classes. Finally, there are a set of miscellaneous tests.

Each of these test sets are discussed in the following sections, but first we discuss the design and organisation of the build rules for invoking these tests.

8.1 Test Design

ESC/Java2 tests are designed to run entirely automatically. There are a number of custom-built test tools that exercise various subsystems including its lexers, parser, etc.

In general, tests run using one of several test harnesses, some of which are specifically designed for a given kind of test, and others are generic jUnit tests.

Each test executes a tool or test harness and generates output. Output from most tools is made generic so that multiple runs will not generate trivially different output. Only the ESC/Java2 tool itself has a special testing mode (see the `-testMode` switch) that ensures that it does not generate trivially different output.

Each test's expected output is stored in a file adjacent to the test source. These files are named either 'ans' in the old SRC ESC/Java test harnesses or in a file of the same name as the test input itself but with a '-expected' suffix in the ESC/Java2 jUnit-based tests.

Adding new tests

8.1.1 The Front-end Test Harnesses

CLASS	TEST_LEXER		**Part:** 1/1
PURPOSE		**INDEXING**	
Tokenizes standard input and prints the resulting stream, one token per output line.		**author:** Joseph Kiniry **cluster:** JAVAFE.PARSER.TEST **keywords:** lexer, scanner, test, front-end	
Inherits from		JAVAFE.PARSER.PRAGMA_PARSER	
Queries		none	
Commands		"Fail with a specific error message.", "Lex a given input stream and output each token on a new line.", "Add the Java keywords to the lexer.", "Add a random lookahead to the lexer.", "Create a test lexer and add a couple of test pragma operators to the lexer.",	
Constraints		"The command-line arguments may only be "javakeywords", "lookahead", or "parsepragmas.""	

Chart 8.1: The Test Harness for the Java Front-end Lexer

```
static_diagram Java_Front-end_Lexer_Test_Harness_Support_Classes
component
  cluster JAVAFE.AST
  component
    deferred class LEXICAL_PRAGMA
    deferred class MODIFIER_PRAGMA
    deferred class STATEMENT_PRAGMA
    deferred class TYPE_DECLARATION_ELEMENT_PRAGMA
  end
  cluster JAVAFE.PARSER
  component
    deferred class PRAGMA_PARSER
  end
  cluster JAVAFE.PARSER.TEST
  component
    effective class TEST_LEXER
    effective class LEXER_TEST_LEXICAL_PRAGMA
    effective class LEXER_TEST_MODIFIER_PRAGMA
    effective class LEXER_TEST_STATEMENT_PRAGMA
    effective class LEXER_TEST_TYPE_DECLARATION_ELEMENT_PRAGMA

    TEST_LEXER inherit PRAGMA_PARSER
    LEXER_TEST_LEXICAL_PRAGMA inherit LEXICAL_PRAGMA
    LEXER_TEST_MODIFIER_PRAGMA inherit MODIFIER_PRAGMA
    LEXER_TEST_STATEMENT_PRAGMA inherit STATEMENT_PRAGMA
    LEXER_TEST_TYPE_DECLARATION_ELEMENT_PRAGMA inherit
      TYPE_DECLARATION_ELEMENT_PRAGMA
  end
end
```

Diagram 8.1: The Test Harness for the Java Front-end Lexer (Support Classes)

CLASS   TEST_PARSER   **Part:** 1/1

PURPOSE   **INDEXING**
 Parse Java compilation units in various ways to   **author:** Joseph Kiniry
test the Java front-end parser.   **cluster:** JAVAFE.PARSER.TEST
                                                    keywords: parser, test, front-end

Queries   "Compare two input streams and prints a message and
                                                returns true if they are different, return false otherwise."

Commands   "Pretty-print a given parsed Java compilation unit to a
                                                specified output stream.",
                                                  "Pretty-print a given parsed Java compilation unit to
                                                the system output stream.",
                                                  "Check the invariant of a parsed Java compilation unit."

Constraints   "If first command-line argument is "diff" then the second
                                                and third command-line arguments must be the name of
                                                the two file to parse and compare as a test.",
                                                  "The command-line arguments may only be "diff", "as-
                                                sert", "check", "print", "progress", "silent", or "idem-
                                                potence".",
                                                  "The command-line argument "diff" may only be the
                                                first command-line argument.",
                                                  "The input stream is expected to contain only Java com-
                                                pilation units."

Chart 8.2: The Test Harness for the Java Front-end Parser

```
static_diagram Java_Front-end_Parser_Test_Harness_Support_Classes
component
  cluster JAVA.IO
  component
    deferred class OUTPUT_STREAM
    deferred class INPUT_STREAM
  end
  cluster JAVAFE.PARSER.TEST
  component
    class TEST_PARSER
    -- Helper classes for the parser test framework.
    effective class MEMORY_PIPE_OUTPUT_STREAM
    effective class MEMORY_PIPE_INPUT_STREAM
    MEMORY_PIPE_OUTPUT_STREAM inherit OUTPUT_STREAM
    MEMORY_PIPE_INPUT_STREAM inherit INPUT_STREAM
  end
end
```

CLASS   TEST_EXPRESSIONS   **Part:** 1/1

PURPOSE	**INDEXING**
Test harness for expression parsing.	**author:** Joseph Kiniry
	cluster: JAVAFE.PARSER.TEST
	keywords: parser, expression, test, front-end
Queries	"Are two Java expressions equivalent, ignoring parentheses?"
Commands	"Fail with a specific error message.", "Parse a given input stream for Java expressions and check the invariant of each parsed expression.", "Parse a given input stream for pairs of comma-separated Java expressions and compare the pairs of parsed expression." "Parse a given input stream for pairs of comma-separated Java expressions, check the invariant of each parsed expressions, and compare the pairs of parsed expression."
Constraints	"The input input stream is expected to contain only Java expressions.", "The input input stream may contain either individual Java expressions, or comma-separated pairs of expressions.", "The command-line arguments may only be "compare" or "check"."

Chart 8.3: The Test Harness for the Java Front-end Expression Parser

8.1.2 The ESC/Java2 Test Harnesses

8.2 Test Execution

'ESCTools/Makefile' 'ESCTools/Javafe/Makefile' 'ESCTools/Escjava/Makefile'

8.3 Java Lexer Tests

'ESCTools/Javafe/test/javafe/test/lex'

8.4 Java Parser Tests

'ESCTools/Javafe/test/javafe/parser/test'

8.5 Java Typechecker Tests

'ESCTools/Javafe/test/javafe/tc/test'

8.6 Java Front-end Tests

'ESCTools/Javafe/test/javafe/test/fe'

8.7 Pragma Parser Tests

'ESCTools/Escjava/test/escjava/parser/test'

8.8 ESC/Java2 Reasoning Tests

'ESCTools/Escjava/test/escjava/parser/test'

8.9 jUnit Tests

'ESCTools/Escjava/test/junittests/'

8.10 JDK Tests

'ESCTools/Escjava/test/jdktests'

8.11 Miscellaneous Tests

'ESCTools/Escjava/test/hofmann/'   'ESCTools/Escjava/test/matrix/'
'ESCTools/Escjava/test/nijmegen/' 'ESCTools/Escjava/test/reasoningbugs/'

List of Charts

List of Diagrams

List of Interface Specifications

Appendix A Copying

Index