RJava User and Developer Manual

Yi Lin
yi.lin@anu.edu.au

November 13, 2013

Abstract

RJava is a restricted subset of the Java language with low-level extensions that allow access to hardware and operating system. RJava utilizes the same syntax as Java, and consequently inherits benefits from Java such as type safety, various software engineering tools and productivity. Futhermore, by restrictions, RJava is a fully static language with closed world assumption. Thus it requires a much more succinct runtime, and is well suitable for aggressive static compilation and optimizations. RJava is designed to be an implementation language for virtual machine construction (and more broadly for system programming).

This manual describes the language and its current implementation– the RJava Compiler (RJC). It is intended for RJava users and developers who are willing to contribute. This manual will be maintained to keep pace with the RJC code base.

Contents

1 RJava Basics

This section describes about RJava's restrictions and extensions, which make RJava differentiate from the vanilla Java.

What is RJava?

> Generally speaking, unless what is *restricted* and what is *extended*, the other part of RJava remains the same as Java (syntactically and semantically) . Besides, the restrictions and the extensions are also compliant with Java *syntax*.

Thus, Java compilers and static analyzers can parse RJava code (they are syntactically same). Java runtimes can execute RJava programs as well; However, Java runtimes will not be able to recognize RJava specific restrictions and extension, thus executing RJava with Java runtimes may result in wrong results, which is not encouraged.

RJava is also compliant with existing Java editors and IDEs. When using Java IDE to edit RJava code, extensions and restrictions need to be imported and also try avoid using execution from IDE.

1.1 RJava Restrictions

One important feature about RJava is that all its restrictions are formalized and any RJava code need to declare the restrictions it complies with. The benefit is that both the developers and the RJava compilers will be able to see the declaration and ensure the restrictions are met.

1.1.1 Restriction Rules and Rulesets

RJava restrictions are defined as Java annotations, and they are annotated by `@RestrictionRule (org.rjava.restriction.RestrictionRule)`. The following is how `@NoException` restriction is defined:

```
1 @RestrictionRule
2 public @interface NoException{
3 }
```

Restriction rules can be used to annotate classes or methods, in the same way as Java annotations are used. However, it is *not* encouraged to directly use restriction rules. The preferred way is to form a `@RestrictionRuleset` by a set of chosen `@RestrictionRule`, and annotate a certain scope of code with such `@RestrictionRuleset`.

One example is the MMTk scope and its restriction `@MMTk`. The following ruleset indicates that any code within `@MMTk` needs to obey `@RJavaCore` ruleset (any RJava code need to include this ruleset, it will be described in next subsection) as well as two additional restrictions (`@NoRuntimeAllocation` and `@Uninterruptible`). The declaration is as below:

```
1 @RestrictionRuleset
2
3 @RJavaCore
4 @NoRuntimeAllocation
5 @Uninterruptible
6 public @interface MMTk{
7 }
```

1.1.2 Restriction Model

Restrictions follow those rules:

- Restriction rules and restriction rulesets apply to scopes of classes and methods to declare restrictions, *not* to fields.
- Restriction rules and rulesets may apply to another restriction ruleset, to indicate that the latter ruleset *includes* those rules and rulesets. This may happen iteratively.
- Any RJava class needs to be restricted by `@RJavaCore`, or restricted by a ruleset that (iteratively) includes `@RJavaCore`.
- A scope restricted by a restriction ruleset is restricted by every single restriction rule within that ruleset.
- If a class is restricted by Rule A, all its methods are restricted by Rule A unless the method is restricted otherwise (particularly `@Uninterruptible` vs. `@Interruptible`).

- Restrictions on a class do *not* affect its child classes. Restrictions on a method do *not* affect its overriding methods.
- Restrictions on a class *affect* its nested classes. As a result, those restrictions also *affect* the methods of the nested class.

1.1.3 RJavaCore Ruleset

`@RJavaCore` declares the basic restrictions that any RJava-compliant code need to follow. It declares as:

```
1  @RestrictionRuleset
2
3  @NoDynamicLoading
4  @NoReflection
5  @NoException
6  @NoCastOnMagicType
7  @NoExplicitLibrary
8  @NoEnum
9  public @interface RJavaCore {
10 }
```

Specifically,

- `@NoDynamicLoading`: the code is not allowed to dynamically load classes. To forbid this, `java.lang.Class.forName()` and `java.lang.Classloader.loadClass()` are not allowed. Also inheriting from java.lang.Classloader is also not allowed.
- `@NoReflection`: the code is not allowed to use any reflection-based feature. To forbid this, any method that may return `java.lang.Class` is not allowed. `@NoExplicitLibrary` already forbids the use of `java.lang.reflect` package. Futhermore, `java.lang.Object.getClass()` is not allowed (`.class` syntax is also not allowed, since on bytecode level, it gets translated into `Object.getClass()`.)
- `@NoException`: the code is not allowed to throw exceptions. To forbit this, `throws` in method declarations and also `throw` statement are not

allowed. Catching blocks are allowed in order to maintain correct syntax when using library methods, however, there will be *no* exception thrown and catching blocks will *not* get executed in any case. It is preferred to write such code as below:

```
Object lock = new Object();
try {
  lock.wait();
} catch (InterruptedException ignore) {}
```

- `@NoCastOnMagicType`: the code is not allowed to do any type casting when *at least one* side is RJava magic types. RJava magic types reside in `org.vmmagic.unboxed` package, including 5 unboxed magic types (`Address`, `Extent`, `ObjectReference`, `Offset`, `Word`) and their array counterparts. Allowed type casting for magic types can be done via provided methods in those classes (more will be discussed in the section about extensions).

- `@NoExplicitLibrary`: the code is not allowed to import any java library. However, classes in the `java.lang.*` package are implicitly imported to any Java code, and they are entangled with Java syntax. RJava *allows* the use of `java.lang.*`. Current RJava Compiler has only implemented a subset of the `java.lang.*` package, more complete implementation will be done in the future.

- `@NoEnum`: the code is not allowed to use enumerate type. Enumerate type is forbidden since enabling full features of Java `enum` would highly involve with dynamic behaviors and extensive use of library methods from `java.lang.Enum`. We designed RJava to be a *fully static* language, thus we forbid the use of enumerate type.

These core restrictions define the RJava language. Any valid RJava code needs to declare their compliance with `@RJavaCore`.

By these restrictions, RJava 1) allows closed world assumption, which helps aggressive static compilation, 2) supports little dynamic behaviors and bares very succinct execution runtime, which makes RJava possible to run with limited hardware resources, and 3) is a simple yet still expressive language.

1.1.4 Other Predefined Restrictions

RJava defines the above *restriction model*. RJava encourages its users to utilize this model and, if favorable, add their own restriction rules and rulesets to more precisely describe restrictions on their own scope. However, RJava predefined a set of restrictions. Some are already mentioned in the sections above. These predefined restrictions are subject to change, check `org.rjava.restriction.rules` package in the code base for the latest information.

1.2 RJava Extensions

In order to undertake system programming task, RJava introduces extensions to allow efficient access to hardware and operating systems. RJava extensions include 1) the `org.vmmagic` package (by Daniel et al.) that allows memory/address representation and operations, and 2) the `org.rjava.osext` package that provides access to operating systems (including some system calls).

1.2.1 org.vmmagic

The `org.vmmagic` package provides 5 unboxed magic types to describe pointer-alike types and their operations. Though implementation may vary, 'unboxed' means those types are not normal RJava objects, and they are more like primitive types though there are methods declared for each type. These magic types share the same length as the pointer length on the target machine:

- `ObjectReference`: can be cast from and to an RJava object. *No* arithmetic, load/store or comparison operations are supported. `ObjectReference` can be cast to `Address` for further operations (unsafe).
- `Address`: used as a pointer-alike type. Arithmetic, load/store and comparison operations are supported.
- `Extent`: used to describe size in bytes (unsigned positive value) Arithmetic and comparison operations are supported.
- `Offset`: used to describe offset in bytes (signed). Arithmetic and comparison oeprations are supported.

- `Word`: used as a pointer-sized integer. Arithmetic, comparison and bitwised operations are supported. Can be cast from and to the other 4 magic types.

The above unboxed magic types each have an array counterpart, namely `ObjectReferenceArray`, `AddressArray`, `ExtentArray`, `OffsetArray`, `WordArray`. They provide `create()`, `get()/set()` and `length()` operations. Arrays of those magic types should *only* be created by using these types.

Besides magic types, `org.vmmagic` provides 'pragma' for supplying information to the compiler. Useful 'pragmas' include `@Inline`, `@NoInline`, `@NoBoundsCheck`, `@NoNullCheck`, etc. Some 'pragmas' from `org.vmmagic` are now considered as an RJava `@RestrictionRule` such as `@Uninterruptible`, and some are very specific to Java and not appliable for RJava thus will be deleted. This part is still a draft, and subject to change.

1.2.2 org.rjava.osext

The `osext` extension allows access to operating systems. Currently `osext` only includes a minimum set of methods for implementing a memory manager (MMTk), and it *will* expand during further development.

- `OSConcurrency`: concurrency/threading related methods.
  - `void mutexLock/Unlock(Object lock)`: provides an alternative to using synchronization on an RJava object.
  - `void threadSuspend/Resume(Thread t)`: suspending an RJava thread (unsafe)
- `OSMemory`: memory related methods.
  - `Address malloc(int size)`: allocates raw memory
  - `Address mmap(Address start, Extent length, int protection, int flags, int fd, Offset offset)`: calls mmap system call
  - `int mprotect(Address start, Extent length, int prot)`: calls mprotect system call
  - `Address memset(Address start, int c, Extent length)`: calls memset system call
- `OSNative`: other methods that calls to native code

- `int errno()`: a wrap of errno() in errno.h
- `String strErrno()`: a warp of strerror()
- `double random()`: generates a random double between 0 and 1

1.3 Relation between RJava and MMTk/JikesRVM

The RJava project was motivated by trying improve the portability of Memory Management ToolKit (MMTk). MMTk started with JikesRVM project, and serves as its memory manager. The same as the rest part of JikesRVM, MMTk is also written in a variant of Java (ad-hoc restrictions with `org.vmmagic` extensions). Though MMTk was designed to be a portable language-agnostic memory manager, its portability was never a success.

One of the main reasons that constrain its portability is the portability of the language it is written in. On one hand, such Java variant is specially tailored, and requires support from its hosting runtime. It is no longer a 'write once run everywhere' Java program, and it cannot execute on a stock Java VM. Whoever wants to host MMTk needs to find a way to execute the Java variant that MMTk is written in. On the other hand, MMTk is written in such an Java variant, and there is an inneglectable performance cost to integrate MMTk(Java) with a hosting runtime written in C/C++ which is the most common case. Past experiences (VMKit, Rotor, GHC) of porting MMTk took an approach of ahead-of-time (AOT) compiling MMTk to native codes, which overcame these two obstacles. However, *none* of their approaches is general or reusable, and each of them took a great effort. Furthermore, in term of performance, *none* of the experiences was reported as a success (meanwhile MMTk hosted by JikesRVM achieves excellent performance).

Thus the idea of formalizing the Java variant that MMTk is written in and providing an AOT compiler to lower such language for effective integration with low-level C/C++ code becomes a promising solution to MMTk portability issues. This is one part of the motivations.

Another important motivation that evolves the RJava project is the clear benefits from using a higher-level language for virtual machine implementation. Compared with traditional approach of using C/C++, higher-level languages provide benefits in safety and productivity, both of which are valuable in VM construction. However, existing experiences of using high-level languages for VM construction (including JikesRVM and MMTk) all formed

their own variant of such high-level languages, which not only results in inability of code reusing but also practically introduces metacircularity issues. As we already found out, metacircularity should be orthogonal to using high-level languages; however, in practice so far, whenever high-level languages are used in virtual machine construction, the chosen implementation language is always the target language of the VM. Thus this project not only wants to settle the reusability/portability issues but also wants to create a general implementation language that can deliver high-level language benefits without falling into metacircular traps.

In short, RJava originates from the ad-hoc coding pattern used in MMTk, and absorbs the `org.vmmagic` package which is also used in MMTk. However, it evolves to be a more general implementation languages to deliver higher-level language benefits as well as good performance.

In next section, we wil describe the usage of the RJava Compiler, current implementation of RJava.

2 RJava Compiler Tools

The RJava Compiler (RJC) is an ahead-of-time compiler for RJava. It parses RJava programs, checks restriction compliance and translates into the target C code. Then the C backends could further compile the generated code into binary or other form of instructions (e.g. LLVM IR).

2.1 Building RJC

The RJava Compiler is written in Java. An ant build file is provided for automatic building.

1. Go to RJC root directory ($root).
2. Use
   ```
   ant -f mybuild.xml
   ```
   After the building succeeds, in $root/build, the following files can be found:
   - components/: a copy of external Java archives that RJC uses;
   - rjava_ext/ : a copy of RJava extensions source file;
   - rjava_rt/ : a copy of RJava runtime source file;
   - rjc.jar: the executable RJava Compiler archive.

2.2 RJava Helloworld

In this subsection, we show an example of how to compiling the simplest RJava hello world into executable.

1. Type in the source code as below, save it as 'HelloWorld.java'. For simplicity, we put the source code in the same directory as rjc.jar (under $root/build). Note that RJava uses the same file name extension as Java (.java).

```
1 // HelloWorld.java
2 import org.rjava.restriction.ruleset.RJavaCore;
3
4 @RJavaCore
5 public class HelloWorld {
```

```
6   public static void main(String[] args) {
7     System.out.println("Hello RJava");
8   }
9 }
```

2. Compiling RJava into C.

```
java -cp rjc.jar:rjava_rt/:. org.rjava.compiler.RJavaCompiler
-rjava_ext rjava_ext/
-rjava_rt rjava_rt/
-soot_jdk components/soot/
-dir .
HelloWorld.java
```

This will invoke RJava Compiler (the arguments passed to RJC will be discussed later, a script is also provided to simplify arguments needed). By default, the generated code locates in output directory of current directory. The RJava runtime sources will also be copied to the output directory, so they can be compiled along with the generated sources. And a GNU Makefile will be generated for compiling C into binary.

3. Compiling C into binary.

```
cd output; make
```

By default, the Makefile will use 'gcc' as the C backend to compile the generated C files. The output binary in this case will be 'HelloWorld' (named after the class where main method is found).

2.3 Full Command Line Options

The basic use of the RJava Compiler is

```
java -cp rjc.jar:rjava_rt/:. org.rjava.compiler.RJavaCompiler
[-option value] [source file(s)]...
```

Here lists all the command line options of the RJava Compiler.

Environment setting options

(Note: the'rjc' script will free you from setting these options manually.)

- **-rjava_ext PATH [REQUIRED]**
  set PATH as the directory for RJava extension source files. PATH should contain org/rjava/osext, org/rjava/restriction and org/vmmagic. RJC requires this option to locate those extensions during parsing RJava sources.

- **-rjava_rt PATH [REQUIRED]**
  set PATH as the directory for RJava runtime source files. PATH should contain the implementation of `java.lang.*` and other runtime source files. RJC will copy the contents in this directory to the output directory, allowing the runtime to be compiled and linked with RJava programs.

- **-soot_jdk PATH [REQUIRED]**
  set PATH as the directory for the JDK jars that Soot needs. PATH should contain two JDK jars, jce.jar and rt.jar. RJC uses Soot to parse RJava source files, and Soot requires those two jars to locate Java's library.

Source file options

- **-dir PATH1:PATH2:... [REQUIRED]**
  set PATH as the classpath of RJava classes; several paths are separated by colon. RJC needs this option to correctly locate RJava files and recognize their packages. For example, org.yourcomp.HelloWorld should be located at PATH/org/yourcomp/HelloWorld.java.

- **SOURCE_FILE(S) [OPTIONAL]**
  name one SOURCE_FILE or several SOURCE_FILES to be compiled. RJC will also compile any classes referenced and used. When left blank, RJC will compile all the RJava files under the defined 'dir'.

- **-l SOURCE_LIST_FILE [OPTIONAL]**
  name a SOURCE_LIST_FILE. A SOURCE_LIST_FILE should be a pure ascii file, and have one source file name per line. RJC will compile all the denoted source files in the SOURCE_LIST_FILE.

Output options

- **-o OUTPUT [OPTIONAL]**
  the final binary will be named as OUTPUT. This option will affect the GNU Makefile that RJC generates. *By default*, OUTPUT is named after the class name which the main method is located in. When RJC meets several main methods during compilation, OUTPUT will be set to the first class that contains a main method, and RJC will report warnings for the following encounters of main methods. When there is no main methods in the source files (usually when compiling a library), OUTPUT will be set as 'lib'.

- **-outdir OUTDIR [OPTIONAL]**
  the target files will be generated to OUTDIR. *By default*, OUTDIR is the 'output' directory under current working directory. This option is currently *ignored*.

- **-m [OPTIONAL]**
  setting this flag will mute console output of RJC. It is usually set when chaining a large number of compilations (e.g. unit testing). *By default*, this flag is unset.

- **-dt [OPTIONAL]**
  setting this flag will facilitate debugging on target code. When C is the target, this flag will additionally set '-g' as gcc flags when generating GNU Makefile. *By default*, this flag is unset.

Target options

- **-m32 [OPTIONAL]**
  setting this flag will instruct the target code to be compiled for 32 bits address. When C is the target, this flag will set '-m32' as gcc flags when generating GNU Makefile. *By default*, this flag is unset.

- **-host_os OS [OPTIONAL]**
  the target code will be compiled for and execute on the named OS. Part of RJava runtime contains OS-dependent code. Currently only MAC OS X ('mac') and Linux ('linux') are supported. *By default*, host OS is 'mac'.

Target-specific options

The RJC allows different targets for RJava to compile into. Options that start with '-target' is considered as target-specific options, and will be processed by different target code generators.

Target-specific options for targeting C

- **-target:mm=VAL [OPTIONAL]**
  name VAL as the memory management scheme for the target code. Currently, available options are using hans-boehm conservative GC ('boehm'), and using default malloc ('malloc'). The latter makes the generated code leak memory since RJava does not provide mechanisms to explicitly free memory; however, it can be useful when allocated memory tends to be permanent. A third option is provided as 'boehm-prebuilt', which uses the static library built beforehand from hans-boehm GC (currently 4 versions of the static libraries exist, 32/64bits version for Linux/Mac). This option is intended to save the time of building hans-boehm GC from sources during testing. *By default*, 'boehm-prebuilt' is used.

2.4 The 'rjc' script

Since setting environment path options for the RJava Compiler is tedious and error-prone, and using RJC jar when current working directory is not the RJC root directory makes the situation even more frustrated, a 'rjc' perl script is provided under the root directory for improved usability.

The 'rjc' script will automatically set those environment paths, and invoke the RJava Compiler under $root/build directory. If the build directory does not exist, it will build the compiler first. All options other than environment setting options can be passed to the script, which will relay to the RJava Compiler. The script also works when current working directory is not the RJC root directory. Be sure to keep the 'rjc' script *unmoved* in the RJC root directory.

It is always preferred to use the 'rjc' script rather than directly using the RJC jar.

In next section, we will describe the implementation of the RJava Compiler, which is intended for RJava developers.

3 RJava Compiler Implementation

The RJava Compiler is an RJava ahead-of-time (AOT) compiler written in Java. Currently it targets only C, translating RJava into C. However, its design bares multiple targets, and more targets can be added in the future while reusing most of the compiler features.

The RJava Compiler achieves the following major tasks:

- utilizing Soot (McGill Sable Group) to parse RJava source files and generating Jimple AST
- performing RJava-level analysis and optimizations (leaving target-level optimizations to backends)
- generating target code and runtime

3.1 Detailed Workflow

This subsection describes the detailed procedure of translating RJava into target code. For easy matching the description here with the actual code, we also reference the key classes and methods.

1. **Processing arguments and forming compilation task**
   The main method of RJava Compiler (`org.rjava.compiler.RJavaCompiler.main()`) starts by processing the arguments and setting options/flags one by one. The source files to compile will be extracted and encapsulated as a CompilationTask (`org.rjava.compiler.CompilationTask`) no matter they are denoted as a file of source list, a set of source file names, or all sources under a certain directory. CompilationTask store source class names and the classpath of the classes (fed as the '-dir' argument).

2. **Instantiating compiler instance**
   A RJava Compiler instance will be instantiated, whose constructor will take the CompilationTask as its argument. After the instantiation of a `org.rjava.compiler.RJavaCompiler` instance (singleton pattern), `init()`, `compile()`, and `finish()` will be executed sequentially. Each of them will be discussed later.

3. **Initializing the compiler**
   The source code parsing (done by Soot) and all the static analyses happen at this step.

(a) **Initializing code generator**
CodeGenerator (`org.rjava.compiler.target.CodeGenerator`, abstract class) is the part to generate the target code. It needs to be initialized according to some arguments processed earlier (`CodeGenerator.init()`).

(b) **Initializing SemanticMap**
SemanticMap (`org.rjava.compiler.semantics.SemanticMap`) stores semantic information about current compiling program and different analyses. This step accomplishes the following tasks:

- **Parsing sources and generating AST**
  Soot is wrapped as `org.rjava.compiler.semantics.SootEngine`. In this step, Soot parse all the related classes (including the imported classes) and add them to current `CompilationTask`. Note that we do not actually 'run' Soot and its optimizations, we simply use Soot to parse sources and use its Jimple AST. *All* the analyses and compilation in RJava Compiler are based on Jimple AST.
- **Instantiating RJC class representation**
  After last step, we have Soot representation of classes (`soot.SootClass`) /methods (`soot.SootMethod`)/etc. To make analyses convenient, we wrap those into RJC representations (see package `org.rjava.compiler.semantics.representation.*`). We only make our own representations for classes, methods, statements and invoke expressions; for the rest, we use Soot representations. Classes that are not denoted as source files but are referenced will also have their RJC representations and RJC will generate code for them (as long as they are application classes, see `SemanticMap.isApplicationClass()`).
- **Removing generic bridging methods**
  Soot may generate bridge methods for generics, which are two methods that have same signature but different return types. One has the actual method body, and the other simply calls that method. In RJC code base, these methods are referred as 'twin methods' (see `org.rjava.compiler.semantics.representation.RMethod` for more details) . We do not need bridging methods, and methods with the same signature complicates name mangling. Twin methods are merged into one whose signature is the

same as defined in source code, and the bridging method is removed.

- **Running multiple analyses**
  All static analyses that RJC needs are done here. The implementation of analyses inherits from CompilationPass (`org.rjava.compiler.pass.CompilationPass`). Note that one analysis may do several passes internally. The analyses include class hierarchy analysis (`ClassHierarchyPass`), call graph (`CallGraphPass`), class initialization dependency (`DependencyGraphPass`), points-to analysis (`PointsToAnalysisPass`), constant propagation (`ConstantPropagationPass`). In addition, a restriction pass (`RestrictionPass`) is executed here to apply RJava's restriction model (e.g. propagating restrictions to nested classes). The results of those analyses can be retrieved through `SemanticMap`.

4. **Compiling**
   After going through the steps above, RJC has parsed source code into AST, and also collected sufficient analysis results. The actual compilation starts from `RJavaCompiler.compile()`. RJC iterates through all the classes in current compilation task, checks their restriction compliance and starts translation. There are also some global target-specific works that need to be done before and after the translation such as clearing output directory and copy runtime sources, and respectively they are done in `CodeGenerator.preTranslationWork()` and `CodeGenerator.postTranslationWork()`.

   (a) **Performing pre-translation work**
   In this step, RJC creates the output directory if it does not exist and clear the directory. This step is 'global' and execute only once while following steps are performed for every RJava class.

   (b) **Checking restrictions**
   RJC checks the restriction compliance of a class before starting to translate it. `org.rjava.compiler.restriction.StaticRestrictionChecker` takes charge of the compliance checking. `comply()` is invoked with an `RClass` argument. For every restriction `org.rjava.restriction.rules.A` declared on such the class, a checking rule (`org.rjava.restriction.rules.A_CHECK`) is expected with a `checkClass()` and `checkMethod()` declared. The

restriction checker will invoke those checking methods. Code violating restrictions, failing to find checking rules or a class not declaring a restriction ruleset will cause an RJava warning (it will not stop the translating process), and will be reported. Details about restriction checking will be covered in next section. Note that current implementation *ignores* restrictions on methods, and does not check them (this will be fixed soon).

(c) **Translating**
This step is target-specific. `CodeGenerator.translate()` is called for the actual code tranlation. `org.rjava.compiler.target.CodeGenerator` is expected to be inherited for a certain target. Currently only C target is implemented. Details about C target translation will be discussed in next section.

(d) **Performing post-translation work** Similar to pre-translation work, post-translation work (`CodeGenerator.postTranslationWork()`) is a target-specific global step. Works such as generating and copying runtime sources, generating build files for auto tools should be done here.

3.2 Codebase Overview

This part describes the overview of RJava Compiler code base, including its main compiler project, unofficial MMTk project clone and other miscellaneous stuff. Importing the root directory of RJC code base into Eclipse IDE will imports the main compiler project.

Current code base is accessible from ANU squirrel mercurial repository `http://squirrel.anu.edu.au/hg/all/yilin/rjava-prototype/` (authorization required).

- **compiler/**
  The main source directory of RJC.
  - `org.rjava.compiler`
    `RJavaCompiler` main class, `CompilationTask` and constants definitions
  - `org.rjava.compiler.exception`
    RJC warnings and errors

- `org.rjava.compiler.pass`
  different analysis passes
- `org.rjava.compiler.restriction`
  the restriction checker
- `org.rjava.compiler.semantics`
  `SemanticMap`, `SootEngine` and other data structures such as `CallGraph`, `DependencyGraph`, etc.
- `org.rjava.compiler.semantics.representation`
  RJC's representation of class, type, method, field, local variable, and annotation to wrap its Soot counterparts. Additionally, `RRestriction` is defined here.
- `org.rjava.compiler.semantics.representation.stmt`
  RJC's representation of 16 different types of statements, which maps to different types of Jimple statements.
- `org.rjava.compiler.target`
  abstract `CodeGenerator` and `GeneratorOptions` that fits into RJC workflow. Each target that RJava wants to translate into need to inherit these two classes. `CodeStringBuilder` is provided here for better indentation of code output.
- `org.rjava.compiler.target.c`
  C-specific code. `CLanguageGenerator` and `CLangaugeGeneratorOptions` inherits the two abstract classes mentioned above. `CStatements` and `CExpressions` generate code for finer-grained scope. `CIdentifiers` mangles Java names into C name. `Intrinsics` takes care of some C-specific intrinsics, e.g. setting RJava integer types to different lengths of integer types in C.
- `org.rjava.compiler.target.c.runtime`
  Part of the RJava's C runtime and the GNU Makefile for the target code are generated during the compilation, which is done here. Also the magic type implementation is generated by reusing RJC to compile magic type stubs; its implementation can be found in `MagicTypesForC`. There is also an attempt to reuse RJC to compile `java.lang.*` package, but it has been put halt.
- `org.rjava.compiler.util`
  data structures such as `Tree`, utility methods for easier use of

third-party components (Soot and JGraphT). We also provide special set, map and multi-value map for storing `soot.Value`, for its special rule of equality comparison. `Statistics` utilities are also included here for instrumenting compiler code.

- **components/**
  Third party components used by RJC, including common collection and IO libraries from Apache, JGraph and JGraphT for visualizing some results of static analyses, and Soot for parsing RJava source code.

- **document/**
  This user and developer manual, both PDF version and tex source.

- **mmtk_standalone/**
  An unofficial clone of Memory Management ToolKit repository. Minor changes from the JikesRVM MMTk are needed to make it compatible with RJava restrictions. Details will be discussed in next section. This can be imported as a separate Eclipse project.

- **mmtk/**
  The symbolic link of the standalone MMTk (the above directory) along with a minimal client required to run MMTk allocation and GC (named as 'testbed'). This can be imported as another separate Eclipse project.

- **rjava_clib_impl/**
  An attempt to write `java.lang.*` in RJava, and reuse RJC to compile it into C target. It has been stopped.

- **rjava_ext/**
  The source files of RJava extensions, including `org.vmmagic` and `org.rjava.osext`. Those source files are mostly empty stubs, and they exist for source code completeness. RJC ensures they will have actual implementation in the target code.

- **test/**
  RJava implementation of some benchmarks, and also some small test cases for different language features and optimizations.

- **unittest/**
  Unit tests for RJava implementation, including a unit test script ('unittest.pl')

and a set of RJava unit test sources. Current unit test is *not* a thorough set covering all the features of RJava, it will be expanded in the future.

3.3 Unit Tests

RJava Compiler includes a simple framework for unit testing.

3.3.1 Running Unit Tests

Run the unit tests by executing

```
unittest/unittest.pl
```

This script will first compile a lastest version of RJava Compiler, then detects all the source files under unittest/src (but not the abstract base class `org.rjava.unittest.UnitTest`). It iterates through every single unit test, invoke RJC to compile RJava into C, then from C into binary, and execute it. It observes the execution of every compilation step, and also tracks the output of the binary. Execution successes and fails are reported at the end of each unit test, and also at the end of the script. The unit testing script is specific to C target.

3.3.2 Adding New Unit Tests

Adding new unit tests is simple. Create a new RJava source file by extending `org.rjava.unittest.UnitTest`, and save it to unittest/unittest/src in order to be discovered by the unit testing script. Within the new unit test source, write your own test code, and use the `start()` and `check()` methods provided by the base class for result outputs so that the testing script can parse and recognize successes and fails.

4 RJava Compiler Details

This section discusses the details about the implementation of RJava Compiler. Part of this section is *specific* to current RJC implementation that targets C. It aims to help developers understand RJC implementation. We will not emphasize this point again in this section.

4.1 RJC Object Model

RJava does not restrict on the use of object-oriented features of Java, and it supports full features of class inheritance and interfaces. To translate the object oriented features into non-OO target namely C, RJC uses an object model similar to GtkObject of GNOME project.

4.1.1 Type Implementation: `RJava_Common_Class`

RJC defines a common type struct, i.e. `RJava_Common_Class`. Every RJava class is matched to a type struct that embeds the common type struct.

The common type struct (defined in `rjava_crt.h`, generated at compile-time):

```
1 struct RJava_Common_Class {
2   RJava_Common_Class* super_class;
3   RJava_Interface_Node* interfaces;
4   pthread_mutex_t class_mutex;
5 };
```

For Java's common `java.lang.Object`, its type struct is:

```
1 struct java_lang_Object_class {
2   RJava_Common_Class class_header;
3
4   // method members
5   struct java_lang_String* (*toString)();
6   ...
7 };
8
9 struct java_lang_Object_class java_lang_Object_class_instance;
```

For a random RJava class `org.yourcomp.A`,

```
package org.yourcomp;

@RJavaCore
public class A {
  int a;
  void foo(int i) {}
}
```

its type struct definition (`org_yourcomp_A.h`) is:

```
struct org_yourcomp_A_class {
  struct java_lang_Object_class header;
  void (*foo_int32_t)(int32_t);
}

struct org_yourcomp_A_class org_yourcomp_A_class_instance;
```

All the class instances will be initialized eagerly in a certain order at the beginning of main method execution (`rjava_class_init()` in `rjava_crt.c`, which is generated at compile-time). Function pointer binding is also done in `rjava_class_init()`.

This design allows a class to access its super class, to declare interfaces and its own method members, and also to selectively override or inherit methods and interface method from its super class.

4.1.2 Object Implementation: `RJava_Common_Instance`

RJC also defines a common instance struct, i.e. `RJava_Common_Instance`. Every RJava object is matched to an instance struct that embeds the common instance struct.

The common instance struct (defined in `rjava_crt.h`, generated at compile-time):

```
struct RJava_Common_Instance {
  void* class_struct;
  pthread_mutex_t instance_mutex;
  pthread_cond_t instance_cond;
};
```

For Java's common `java.lang.Object`, its instance struct is:

```
struct java_lang_Object {
  RJava_Common_Instance instance_header;

  // instance fields
  ...
};
```

For a random RJava class `org.yourcomp.A` (see the above subsection), its instance struct definition (`org_yourcomp_A.h`) is:

```
struct org_yourcomp_A {
  java_lang_Object instance_header;

  int32_t a;
}
```

This design allows every instance to access its type information, and to inherit fields declared by its super class.

4.1.3 Interface Implementation: `RJava_Interface_Node`

An interface class in RJava is translated into a type struct and an instance struct in the same way as normal classes. Interfaces are instantiated for each class that implements them. For accessing interface instances from a type, `RJava_Interface_Node` is used as a linked list to store all the interface instances declared on this type:

```
struct RJava_Interface_Node {
```

```
  char* name;
  void* address; // points to interface instance
  int interface_size;
  RJava_Interface_Node* next;
};
```

The instantiation and initialization of interface instances and `RJava_Interface_Node` for each type is done in `rjava_class_init()`. RJC's C runtime provides methods to help with the initialization and also with interface invocation:

- `void rjava_add_interface_to_class(void* interface, int interface_size, char* name, RJava_Common_Class* klass);`
- `void rjava_alter_interface(void* interface, char* name, RJava_Common_Class* klass);`
- `void* rjava_get_interface(RJava_Interface_Node* list, char* name);`

Note that current implementation of interface makes `invokeinterface` *extremely inefficient*, and we have not implemented any optimizations for interface. Use of interfaces should be discouraged in performance-centric scenarios until the implementation gets an improvement.

4.1.4 Array Implementation

RJava arrays is implemented different than Java arrays. RJava arrays is *not* an `Object`, and it is more like C arrays with a header. The header is required to perform bounds checking. For the address that an array points to, first thing accessed is a C integer as the array length, followed by a C long as element size. The rest is the actual array.

This implementation is flawed, since an array does not carry its type information and prevents run-time type checking. Besides, not being an RJava `Object` is conceptually different than Java, which is not desired. The array implementation will be revised.

4.2 Magic/Unboxed Types

Magic types are pointer-sized integers that are used to describe memory related operands. In RJava source code, they are expressed as normal RJava

classes; however, they are intended to behave more like 'primitives' rather than 'objects'.

In RJC implementation, uses of magic types in RJava code are translated in the same way as other types (though RJC understands those are magic types). And invocations for methods in magic types are translated into a static function call (normal RJava application method invocation is translated a function call via function pointer unless de-virtualized).

RJC provides an implementation of magic types and their methods as a large list of C macros. So during C compilation, magic types and their method callsites will be replaced. Magic types are each defined as:

```
#define org_vmmagic_unboxed_Address       uintptr_t
#define org_vmmagic_unboxed_Extent        uintptr_t
#define org_vmmagic_unboxed_ObjectReference  uintptr_t
#define org_vmmagic_unboxed_Offset        intptr_t
#define org_vmmagic_unboxed_Word          uintptr_t

#define org_vmmagic_unboxed_AddressArray   uintptr_t*
#define org_vmmagic_unboxed_ExtentArray    uintptr_t*
#define org_vmmagic_unboxed_ObjectReferenceArray  uintptr_t*
#define org_vmmagic_unboxed_OffsetArray      intptr_t*
#define org_vmmagic_unboxed_WordArray        uintptr_t*
```

Their methods are defined as (only showing a few as examples):

```
#define
    org_vmmagic_unboxed_Address_plus_int32_t(this_parameter,parameter0)
    \
this_parameter + parameter0
#define
    org_vmmagic_unboxed_Address_plus_org_vmmagic_unboxed_Offset(this_parameter,parameter
    \
this_parameter + parameter0
```

This implementation requires minimal special treatment on magic types during translation, and brings in low run-time overhead. The implementation of magic types can be generated by reusing RJC

(see `org.rjava.compiler.target.c.runtime.MagicTypesForC`).

4.3 `java.lang.*` Package

RJava disallows the use of Java library by `@NoExplicitLibrary` restriction for its own simplicity. However, `java.lang.*` is entangled with the Java syntax and it is hardly possible to forbid its use without crippling the language. Thus RJava preserves `java.lang.*`.

RJC implements the package in C by following its object model. There is only one difference between RJC`java.lang.*` and an RJava program that method invocations to `java.lang.*` are statically bounded. This implies that it is *not* allowed to inherit from the library classes.

RJC selectively picks a subset of `java.lang.*` to implement, following these principles:

- enabling primitive type boxing
- providing string type
- providing support for threading and concurrency
- providing empty stubs just for source completeness
- providing basic standard output

Thus current RJC includes *incomplete* implementation of the following classes:

- `java.io.PrintStream` (only for standard output)
- `java.lang.Boolean`
- `java.lang.Class` (empty stub)
- `java.lang.ClassNotFoundException` (empty stub)
- `java.lang.Double`
- `java.lang.Exception` (empty stub)
- `java.lang.Float`

- `java.lang.Integer`
- `java.lang.InterruptedExeption` (empty stub)
- `java.lang.Math`
- `java.lang.NoClassDefFoundError` (empty stub)
- `java.lang.Object`
- `java.lang.Runnable`
- `java.lang.String`
- `java.lang.StringBuffer`
- `java.lang.System`
- `java.lang.Thread`
- `java.lang.Throwable` (empty stub)

Current design and implementation about `java.lang` remains as a draft, and it might need revision in the future.

4.4 RJava Compiler AST

RJC uses Soot to parse source code, and consequently also reuses Soot's Jimple AST. Jimple's grammar is showed below (quoting Raja Vallee-Rai's thesis):

- $stmt \longrightarrow assignStmt \mid identityStmt \mid gotoStmt \mid$
  $ifStmt \mid invokeStmt \mid switchStmt \mid$
  $monitorStmt \mid returnStmt \mid throwStmt \mid$
  $breakpointStmt \mid nopStmt$
- $assignStmt \longrightarrow \mathbf{local} = \mathbf{rvalue};\ \mid$
  $\mathbf{field} = \mathbf{imm};\ \mid$
  $\mathbf{local}.\mathbf{field} = \mathbf{imm};\ \mid$
  $\mathbf{local}[\mathbf{imm}] = \mathbf{imm};$

- *identityStmt* $\longrightarrow$ **local** = **this**; |
  local = **parameter$_\mathbf{n}$**; |
  ~~**local = exception**;~~
- *gotoStmt* $\longrightarrow$ *goto label*;
- *ifStmt* $\longrightarrow$ *if* **conditionExpr** *goto label*;
- *invokeStmt* $\longrightarrow$ *invoke* **invokeExpr**;
- *switchStmt* $\longrightarrow$ *lookupswitch* **imm**
  {*case* **value$_\mathbf{1}$** : *goto* $label_1$;
  ...
  case **value$_\mathbf{n}$** : *goto* $label_n$;
  default : *goto defaultLabel*; }; |
  tableswitch **imm**
  {*case* **low** : *goto lowLabel*;
  ...
  case **high** : *goto highLabel*;
  default : *goto defaultLabel*; };
- *monitorStmt* $\longrightarrow$ *entermonitor* **imm**; |
  exitmonitor **imm**;
- *returnStmt* $\longrightarrow$ *return***imm** |
  return;
- ~~*throwStmt* $\longrightarrow$ *throw* **imm**~~
- ~~*breakpointStmt* $\longrightarrow$ *breakpoint*;~~
- *nopStmt* $\longrightarrow$ *nop*;

Note:

1. **bold font** means a *Value* type (`soot.Value`)
2. ~~strikeout font~~ means rules that are not appliable to RJava.

3. RJC implements wrapper classes to represent different Jimple statements (see `org.rjava.compiler.semantics.representation.stmt`), however, for *Value* and subclasses of *Value*, RJC directly use Soot representations.

4.5 Analysis and Optimization passes

This subsection gives detailed information about some analysis and optimization passes in RJC by describing them or referencing the origin of the algorithm. All analyses so far (inherited from `org.rjava.compiler.pass.CompilationPass`) are *flow-sensitive* and *path-insensitive*.

4.5.1 Class Initialization Order

RJC ahead-of-time compiles RJava code, and forbids any run-time class loading. Thus all the classes can be seen at compile-time, and early class initialization is desired.

RJC uses the algorithm from the paper 'Eager Class Initialization for Java' by Dexter Kozen and Matt Stillerman. The algorithm generates a class dependency graph; each edge A→B means the initialization of class A depends on the initialization of class B, and as a result, B should be initialized before A.

4.5.2 Constant Propagation

Though the C backend will do constant propagation, during translation from RJava to C, some high-level information is lost which makes C compiler easily miss opportunities.

RJC uses the algorithm from the paper 'Interprocedural Constant Propagation' by David Callaham et al. RJC only considers *numeric* constants (including integer, floating point number, byte and character. Address-alike constants are currently ignored.)

This optimization pass does two passes over the code to collect information, and performs the propagation in the end. It uses results from *Class Hierarchy Analysis (CHA)*, *Points-to Analysis (PTA)* and *Call Graph (CG)*. Note this optimization does not directly change AST, but store constant information internally and provide query methods for use at code generation time.

This optimization is optimistic, and specially takes care of situations such as:

- numeric `static final` fields used across different RJava files should be deduced as constant if possible.
- methods that returns constant should be propagated.
- methods that take same constant arguments from all their callsites can be propagated.
- methods that might be called by polymorphism and also match the two points above can be propagated.

4.5.3 Devirtualization

Devirtualization is the most crucial optimization in RJC in term of performance. A virtualized method invocation of `obj.methodFoo()` (`obj` is an instance of `ClassFoo`) under RJC object model is `((ClassFoo_class*)(((RJava_Common_Instance*)obj)-> class_struct))-> methodFoo()`, and a devirtualized version can be `ClassFoo_methodFoo()`. This not only eliminates the pointer dereferencing, but also allows inlining from the C backend. This optimization boosts the performance by a factor of 3 when running MMTk.

Devirtualization is not a `CompilationPass` in RJC. It uses the results from points-to analysis for type inference when generating code for virtual invocations.

4.6 Restriction Checking

Checking the compliance of restrictions is important for RJava code robustness. RJC checks the restriction at compile-time.

RJC assume every restriction rule (`org.rjava.restriction.rules.A`) comes with a checking rule (`org.rjava.restriction.rules.A_CHECK`), and the checking class should contain methods with signature:

- `public static Boolean checkClass(RClass klass);`
- `public static Boolean checkMethod(RMethod method);` (this one is not currently implemented nor invoked, RJC only checks classes at the moment)

The implementation of checking methods is supposed to:

- return true if the checking passes, or false if the checking fails.
- create an `RJavaRestrictionViolation` object by `RJavaRestrictionViolation.newRestrictionViolation()` if checking fails, which stores information about this violation.
- inform the `StaticRestrictionChecker` of the violation by `StaticRestrictionChecker.addRestrictionViolation()`.

RJC's restriction checker will invoke such methods from these classes for every declared restriction rules. If a class fails the restriction checking (either violating the restriction or failing to find checking classes/methods), a warning will be recorded and reported at the end of compilation (it will not interrupt compilation).

4.7 C Target Translation

The `org.rjava.compiler.target.c` package deals with the C target translation. Ideally this should be a package dedicated to C target, which means code inside should be C specific and not general while code outside should be general and target independent. Current implementation roughly follows this rule (however, not *strictly*); further refactoring will fix this.

The generated code currently complies with GNU99 standard. We may move to standard C99 in the future.

4.7.1 Code Generation

`CLanguageGenerator` is responsible for code generation with the help from `CStatements`, `CExpressions` and `CIdentifiers`.

- A non-interface RJava class `org.yourcomp.A` will be translated into 3 different C files:
  1. a header file `org_yourcomp_A.h`, which contains the definition of the type '`struct org_yourcomp_A_class`' and the instance '`struct org_yourcomp_A`' from this class,

2. a header file `org_yourcomp_A_methods.h`, which contains the declaration of methods from `org.yourcomp.A` and the definition of inlining methods,
3. a source file `org_yourcomp_A_methods.c`, which contins the definition of non-inlining methods.

Every C file will include the type or methods it used. The separation of method definition and type/instance definition can to the maximum eliminate possible circular referencing between two different header files.

- An RJava interface class `org.yourcomp.InterfaceA` will be translated into 2 different C files:

  1. a header file `org_yourcomp_InterfaceA.h`, which contains the definition of the type '`struct org_yourcomp_InterfaceA_class`', and the instance '`struct org_yourcomp_InterfaceA`' from this class,
  2. a source file `org_yourcomp_InterfaceA.c`, which *only* contains a `<clinit>` method if exist.

- During the code generation for an RJava class, the following information will be collected:

  1. types and methods that are referenced. This will instruct the generation of an `include` list.
  2. type header and implemented interface initialization code. This consists the `rjava_class_init()` in addition to Java's `<clinit>`.

- Name mangling in RJC follows the following rules (see `CIdentifiers`):

  – **package name and class name**: dot is replaced by underscore. E.g. `org.yourcomp.A` → `org_yourcomp_A`.

  – **method name**: `<init>` → `rjinit`, `<clinit>` → `rjclinit`, others remain the same.

  – **method signature in definition**: package name, class name, method name, parameter types, concatenated with underscore.
  E.g. `void org.yourcomp.A.foo(int)` → `void org_yourcomp_A_foo_int32_t(int)`.

  - **function pointer in type header**: method name and parameter types, concatenated with underscore.
    E.g. `void org.yourcomp.A.foo(int)` → `void (*foo_int32_t)(int)`.
  - **static field**: 'var', class name, field name, concatenated with underscore. (adding 'var' to avoid clash with function names)
    E.g. `int a` in `org.yourcomp.A` → `int var_org_yourcomp_A_a`.
  - **nested class**: the same as Java name mangling. However, dollar symbol is replaced by '_NESTED_' when it appears in file name.
    E.g. `org.yourcomp.A.NestedClass` → `org_yourcomp_A$NestedClass` with file name `org_yourcomp_A_NESTED_NestedClass`.

- Method parameters are named as `parameterN` where N is the index of this parameter. An instance method has its first parameter as a pointer (`void*`) to its receiver, named as `this_parameter`.

- Main method uses C signature of '`int main(int argc, char** parameter0)`', where `parameter0` is a C array and is transformed into an RJava C array at the beginning of main method by using a runtime helper `rjava_c_array_to_rjava_array()`. Before main method returns, all threads will be joined (`rjava_join_all_threads()`).

4.7.2 Runtime

Part of RJC's C runtime is hand coded library implementation written in C, and the other part is generated during code generation, i.e. `rjava_crt.h`. `rjava_crt.h` is included by every C file generated by RJC.

`rjava_crt.h` includes:

- includes of C libraries, such as standard libraries (`stdio.h`, `stdlib.h`, `stdbool.h`, `inttypes.h`, `pthread.h` and `limits.h`) and external C libraries (hans-boehm GC).

- defines, such as OS define (`#define __OS_MACOSX_` or `#define __OS_LINUX_`, type define (`#define byte char`).

- RJC object model, such as `RJava_Common_Class`, `RJava_Common_Instance`, and `RJava_Interface_Node`.

- type declarations of all application types.

- runtime global variables.
- runtime helper functions:
  - `void rjava_class_init();`
  - `void rjava_assert(bool cond, char* str);`
  - `void rjava_add_interface_to_class(void* interface, int interface_size, char* name, RJava_Common_Class* klass);`
  - `void rjava_alter_interface(void* interface, char* name, RJava_Common_Class* klass);`
  - `void* rjava_get_interface(RJava_Interface_Node* list, char* name);`
  - `void rjava_init_header(void* this_class, void* super_class, int super_class_size);`
  - `void rjava_debug_print_header(void* this_class, char* name);`
  - `void* rjava_new_array(int length, long ele_size);`
  - `inline int rjava_length_of_array(void* array);`
  - `inline void* rjava_access_array(void* array, int index);`
  - `inline void* rjava_access_array_nobounds_check(void* array, int index);`
  - `void* rjava_init_args(int argc, char** args);`
  - `void* rjava_new_multiarray(int* dimensions, int dimension_size, long ele_size);`
  - `void* rjava_c_array_to_rjava_array(int length, long ele_size, void* c_array);`
  - `bool rjava_instanceof(void* instance, void* class_struct);`
  - `void rjava_init_thread_suspending();`
  - `void rjava_unimplemented_method();`
  - `void rjava_runtime_global_init();`
- definitions of inlining helper funtions.

`rjava_crt.c` contains the definitions of the other helper functions.

5 MMTk/RJava Manual

MMTk updates with JikesRVM. However, current RJC code base contains an unofficial branch of MMTk. Though MMTk has nothing to do with RJC implementation, MMTk is the most important project that is easily compatible with RJC.

5.1 Unofficial Changes

As described earlier, RJava originates from MMTk but is generalized and formalized. This makes current design of RJava is slightly different that in MMTk. We apply current design to MMTk code, and make a few changes to make it fully compatible with RJC, which is the unofficial branch of MMTk in RJC code base.

Having an unoffical branch is a temporary solution, and we seek to push those changes back to JikesRVM's MMTk. However, this may happen after RJC becomes more robust.

The changes include:

- **`@MMTK` restriction ruleset**
  Every classes in MMTk is annotated with this ruleset.

```
1 @RestrictionRuleset
2
3 @RJavaCore
4 @NoRuntimeAllocation
5 @Uninterruptible
6 public @interface MMTk {
7 }
```

- **Removing uses of `enum`**
  RJava disallows the use of enumerate type. Thus `org.mmtk.utility.statistics.Stats.Phase` and `org.mmtk.vm.ReferenceProcessor.Semantics` are removed, and their use are replaced with an `int`.

- **Removing use of `java.lang.Class`**
  To eliminate any potential dynamic loading and reflection, RJava disallows any use of `java.lang.Class`. Though it is known that MMTk

does not dynamically load classes nor use reflection, MMTk has a few occurrences of `java.lang.Class`. Changs are lised below:

  - `CollectorContext` of each `Plan`. For a `Plan`, its `CollectorContext` used to be calculated from class name, then retrieved by `Class.forName()` and new instances are created from from it. Since those uses are forbidden in RJava, the code is refactorred. The changes involve `org.mmtk.plan.Plan.<clinit>`, `org.mmtk.plan.ParallelCollectorGroup.initGroup()`, and `org.mmtk.plan.Plan.newCollectorContext()` (newly added abstract method).
  - specialized scan class. `org.mmtk.plan.TransitiveClosure` stores specialized scan class for current plan, however, MMTk does not directly use those classes but pass them to the hosting VM. Now MMTk simply stores class names of the specialized scan class, and let the hosting VM do the `Class.forName()` work. Changes involve `org.mmtk.plan.TransitiveClosure` and `org.mmtk.plan.Plan.getSpeicalizedScanClass()`.
  - MMTk-VM interface factory. MMTk provides a set of abstract classes as MMTk-VM interface that the hosting VM has to implement. `org.mmtk.vm.Factory` is responsible to create instances of the actual implementation (this happens in `org.mmtk.vm.VM`). However, the factory class is instantiated from its class name, thus is forbidden. This is resolved by making `org.mmtk.vm.VM` as a host-specific class so that the factory is hard coded by the host VM.

- **Removing GCSpy and GCTrace**
  The use of GCSpy and GCTrace in MMTk is obsolete, and is planned to be removed from its code base. For simpliciy, they are removed from RJC's branch.

5.2 VM-MMTk Interface