Chord: A Versatile Platform for Program Analysis

Mayur Naik

April 15, 2011

Part I

Preface

Chapter 1

Organization of this Guide

Chord is a program analysis platform that is both *stand-alone*, in that it provides many standard analyses for users to run, and *extensible*, in that it allows users to write their own analyses, possibly atop the provided analyses. As a result, Chord has two kinds of users: *end-users*, who only wish to run predefined analyses, and *developers*, who additionally wish to write and run their own analyses.

For convenience, this user guide consists of two parts: a guide for end-users and a guide for developers. Unlike end-users, developers need to understand Chord's source code and API, and code written by them is executed as part of a Chord run. Hence, the guide for end-users concerns how to run Chord, and the guide for developers concerns how to extend Chord.

Chapter 2

Acknowledgments

Chord would not be possible without the following open-source software:

- Joeq, a Java compiler framework
- Javassist, a Java bytecode manipulation framework
- bddbddb, a BDD-based Datalog solver

Chord additionally relies on the following open-source tools and libraries:

- Ant-Contrib, a collection of useful Ant tasks
- BuDDy, a BDD library
- GNU Trove, a primitive collections library for Java
- Java2HTML and Java2Html, Java-to-HTML tools
- Saxon, an XSLT processor

Chord was supported in part by grants from the National Science Foundation, an equipment grant from Intel, and a Microsoft fellowship during 2005-2007.

Part II

Guide for End-Users

Chapter 3

What is Chord?

Chord is a program analysis platform that enables users to productively design, implement, evaluate, and combine a broad variety of static and dynamic program analyses for Java bytecode. It has the following key features:

- It provides various off-the-shelf analyses (e.g., various may-alias and call-graph analyses; thread-escape analysis; static slicing analysis; static and dynamic concurrency analyses for finding races, deadlocks, and atomicity violations; etc.)
- It allows users to express a broad range of analyses, including both static and dynamic analyses, analyses written imperatively in Java or declaratively in Datalog, summary-based as well as cloning-based interprocedural context-sensitive analyses, iterative refinement-based analyses, client-driven analyses, and combined static/dynamic analyses.
- It executes analyses in a demand-driven fashion, caches results computed by each analysis for reuse by other analyses without re-computation, and can execute analyses without dependencies between them in parallel.
- It guarantees that the result is the same regardless of the order in which different analyses are executed; moreover, results can be shared across different runs.

Chord is intended to work on a variety of platforms, including Linux, Windows/Cygwin, and MacOS. It is open-source software distributed under the New BSD License. Improvements by users are welcome and encouraged. The project website is `http://jchord.googlecode.com/`.

Chapter 4

Getting Started

This chapter describes how to download, install, and run Chord. Section 4.1 describes how to obtain pre-built binaries of Chord. Section 4.2 describes how to obtain the source code of Chord and Section 4.2 explains how to build it. Finally, Section 4.4 describes how to run Chord.

4.1 Downloading Binaries

To obtain Chord's pre-built binaries, download and uncompress file chord-bin-2.0.tar.gz. It includes the following files:

1. `chord.jar`, which contains the class files of Chord and of libraries used by Chord.
2. `libbuddy.so`, `buddy.dll`, and `libbuddy.dylib`: you can keep one of these files depending upon whether you intend to run Chord on Linux, Windows/Cygwin, or MacOS, respectively. These files are needed only if you want BDD library BuDDy to be used when the BDD-based Datalog solver bddbddb in Chord runs analyses written in Datalog.
3. `libchord_instr_agent.so`: this file is needed only if you want the JVMTI-based bytecode instrumentation agent to be used when Chord runs dynamic analyses.

Novice users can ignore items (2) and (3) until they become more familiar with Chord. The binaries mentioned in items (2) and (3) might not be compatible with your machine, in which case you can either forgo using them (with hardly any noticeable difference in functionality), or you can download the sources (see Section 4.2) and build them yourself (see Section 4.3).

4.2 Downloading Source Code

To obtain Chord's source code, download and uncompress the following files:

- Mandatory: file chord-src-2.0.tar.gz, which contains Chord's source code and jars of libraries used by Chord.
- Optional: file chord-libsrc-2.0.tar.gz, which contains the source code of libraries used by Chord (e.g., joeq, javassist, bddbddb, etc.)

Alternatively, you can obtain the latest development snapshot from the SVN repository by running the following command:

```
svn checkout http://jchord.googlecode.com/svn/trunk/ chord
```

Instead of checking out the entire `trunk/`, which contains several sub-directories, you can check out specific sub-directories:

- `main/` contains Chord's source code and jars of libraries used by Chord.
- `libsrc/` contains the source code of libraries used by Chord (e.g., joeq, javassist, bddbddb, etc.).
- `test/` contains Chord's regression tests.
- many more; these might eventually move under `main/`.

Files `chord-2.0-src.tar.gz` and `chord-2.0-libsrc.tar.gz` mentioned above are essentially stable releases of the `main/` and `libsrc/` directories, respectively.

4.3 Compiling the Source Code

Compiling Chord's source code requires the following software:

- JVM for Java 5 or higher, e.g. IBM J9 or Oracle HotSpot.
- Apache Ant, a Java build tool.

Chord's main directory contains a file named `build.xml` which is interpreted by Apache Ant. To see the various possible targets, simply run command "`ant`" in that directory.

To compile Chord, run command "`ant compile`" in the same directory. This will compile Chord's Java sources from `src/` to class files in `classes/`, as well as build a jar file `chord.jar` that contains these class files as well as those in the jars of libraries that are used by Chord and are provided under `lib/` (e.g., `joeq.jar`, `javassist.jar`, `bddbddb.jar`, etc.). Additionally:

- If system property `chord.use.buddy` is set to `true`, then the C source code of BDD library BuDDy from directory `bdd/` will be compiled to a shared library named (`libbuddy.so` on Linux, `buddy.dll` on Windows, and `libbuddy.dylib` on MacOS; this library is used by BDD-based Datalog solver bddbddb in Chord for running analyses written in Datalog.
- If system property `chord.use.jvmti` is set to `true`, then the C++ source code of the JVMTI-based bytecode instrumentation agent from directory `agent/` will be compiled to a shared library named `libchord_instr_agent.so` on all architectures; this agent is used in Chord for computing analysis scope dynamically and for running dynamic analyses.

Properties `chord.use.buddy` and `chord.use.jvmti` are defined in a file named `chord.properties` in Chord's main directory. The default value of both these properties is `false`. If you set either of them to `true`, then you will also need a utility like GNU Make (to run the `Makefile`'s in directories `bdd/` and `agent/`) and a C++ compiler (to build the above shared libraries).

4.4 Running Chord

Running Chord requires a JVM supporting Java 5 or higher. There are two equivalent ways to run Chord. One way, which is available in both the source and binary installations of Chord, is to run the following command:

```
java -cp <CHORD_MAIN_DIR>/chord.jar -D<key1>=<val1> ... -D<keyN>=<valN> chord.project.Boot
```

where `<CHORD_MAIN_DIR>` denotes the directory containing file `chord.jar`; that directory is also expected to contain any shared libraries in Chord's installation (e.g., `libbuddy.so` and `libchord_instr_agent.so`).

The alternative, which is available only in the source installation of Chord, is to run the following command:

```
ant -f <CHORD_MAIN_DIR>/build.xml -D<key1>=<val1> ... -D<keyN>=<valN> run
```

This approach requires Apache Ant (a Java build tool) to be installed on your machine. We use this approach throughout this documentation. Also, we omit the "`-f<CHORD_MAIN_DIR>/build.xml`" argument on the command-line for brevity.

Each "`-D<key>=<val>`" argument above sets the system property named `<key>` to the value denoted by `<val>`. The only way to specify inputs to Chord is via system properties; there is no command-line argument processing. Chapter 5 describes all system properties recognized by Chord.

Chapter 5

Chord Properties

The only way to specify inputs to Chord is by means of system properties. There is no command-line argument processing in Chord and any command-line arguments are ignored. Section 5.1 explains how to set properties and Section 5.2 explains the meaning of properties recognized by Chord. Notation `[<...>]` is used in this chapter to denote the value of the property named `<...>`.

5.1 How to Set Properties

A property can be passed to Chord in any of several ways. The reason for providing multiple ways is to provide users with shorthand ways for defining properties once and for all for a particular Java program under analysis, or even once and for all across all Chord runs. The following are the different ways by which a property can be passed to Chord in decreasing order of precedence:

1. **How:** On the command-line via the "`-D<key>=<val>`" format.

   When: Use this option to specify properties specific to the *current run* of Chord.

   Typical usage of this option is by running the following command:

```
ant -D<key1>=<val1> ... -D<keyN>=<valN> run
```

2. **How:** Via a user-defined *properties file* whose location is specified by property `chord.props.file`.

   When: Use this option to specify once and for all properties of the Java program to be analyzed (e.g., property `chord.main.class` specifying the name of that program's main class). Chapter 6 presents an example properties file that defines the program properties that are most commonly used. Section 5.2.1 presents all program properties that are recognized by Chord.

   The default value of property `chord.props.file` is `[chord.work.dir]/chord.properties`. Property `chord.work.dir` specifies the directory in which Chord must run; its default value is the current directory.

There are three ways to use this option of setting properties:

The first way is to override the default value of property `chord.work.dir` on the command-line. This requires naming the above properties file as `chord.properties`, placing it in the directory in which Chord will run, denoted `<WORK_DIR>` (e.g., this could be the top-level directory of the program to be analyzed), and running the following command:

```
ant -Dchord.work.dir=<WORK_DIR> run
```

The second way is to override the default value of property `chord.props.file` on the command-line. In this case, the properties file can be in any user-desired location, denoted `<PROPS_FILE>`, and Chord will run in the current directory:

```
ant -Dchord.props.file=<PROPS_FILE> run
```

The third (and most flexible) way is to override the default values of both properties `chord.work.dir` and `chord.props.file` on the command-line.

3. **How:** Via the properties file named `chord.properties` that is already provided in Chord's main directory.

   When: Use this option to specify once and for all properties you would like to hold in *every run* of Chord (e.g., property `chord.max.heap` specifying the maximum heap memory size to be used by the JVM running Chord).

5.2 Recognized Properties

The following properties are recognized by Chord. The separator for list-valued properties can be either a blank space, a comma, a colon, or a semi-colon.

5.2.1 Java Program Properties

This section describes properties of the Java program to be analyzed, such as its main class, the location(s) of its class files and Java source files, and command-line arguments to be used when running the program.

`chord.work.dir`

Type: location
Description: Working directory during Chord's execution. This is usually the top-level directory of the input Java program.
Default value: current working directory

`chord.props.file`

Type: location
Description: Properties file loaded by Chord at the beginning before doing anything else. Any of the below properties may be defined in this file to avoid defining them on the command line (using the `-D<key>=<val>` format) every time Chord is run. Each relative file/directory name in the value of any property defined in this file is treated relative to Chord's working directory (which is specified by property `chord.work.dir`).
Default value: `[chord.work.dir]/chord.properties`

`chord.main.class`

Type: class
Description: Fully-qualified name of the main class of the input Java program (e.g., `com.example.Main`).

`chord.class.path`

Type: path
Description: Classpath of the input Java program. It does not need to include boot classes (i.e., classes in `[sun.boot.class.path]`) or standard extensions (i.e., classes in jar files in directory `[java.home]/lib/ext/`).
Default value: `""`

`chord.src.path`

Type: path
Description: Source path of the input Java program.
Default value: `""`
Note: Chord analyzes only Java bytecode, not Java source code. This property is used only by the task of converting Java source files into HTML files by analyses that need to present their results at the Java source code level (by calling method `chord.program.Program.g().HTMLizeJavaSrcFiles()`).

`chord.run.ids`

Type: string list
Description: List of IDs to identify runs of the input Java program.
Default value: `0`
Note: This property is used only when Chord runs the input Java program, namely, when it is asked to compute the analysis scope dynamically (i.e., when `[chord.scope.kind]=dynamic`) or when it is asked to run a dynamic analysis.

`chord.args.<id>`

Type: string
Description: Command-line arguments string to be used for the input Java program in the run having ID `<id>`.

Default value: `""`
Note: This property is used only when Chord runs the input Java program, namely, when it is asked to compute the analysis scope dynamically (i.e., when `[chord.scope.kind]=dynamic`) or when it is asked to run a dynamic analysis.

`chord.runtime.jvmargs`

Type: string
Description: Arguments to JVM which runs the input Java program.
Default value: `"-ea -Xmx1024m"`
Note: This property is used only when Chord runs the input Java program, namely, when it is asked to compute the analysis scope dynamically (i.e., when `[chord.scope.kind]=dynamic`) or when it is asked to run a dynamic analysis.

5.2.2 Analysis Scope Properties

This section describes properties that specify how the analysis scope of the input Java program is computed. See Chapter 7 for more details.

`chord.scope.kind`

Type: `[dynamic|rta|cha]`
Description: Algorithm to compute analysis scope. The choices are `dynamic` (dynamic analysis), `rta` (Rapid Type Analysis), and `cha` (Class Hierarchy Analysis).
Default value: `rta`
Note: This property is ignored if property `chord.reuse.scope` is set to `true` and the files specified by properties `chord.methods.file` and `chord.reflect.file` exist.

`chord.reflect.kind`

Type: `[none|dynamic|static|static_cast]`
Description: Algorithm to resolve reflection. The choices are `none` (do not resolve any reflection), `dynamic` (run the program and observe how reflection is resolved), `static` (resolve reflection statically but without analyzing casts), and `static_cast` (resolve reflection statically by analyzing casts).
Default value: `none`

`chord.ch.kind`

Type: `[static|dynamic]`
Description: Algorithm to build the class hierarchy. If it is `dynamic`, then the input Java program is executed and classes not loaded by the JVM while running the program are excluded while building the class hierarchy.
Default value: `static`
Note: This property is relevant only if `chord.scope.kind` is `cha` since only this scope computing algorithm queries the class hierarchy.

`chord.ssa`

Type: bool
Description: Do SSA (Static Single Assignment) transformation of the bodies of all methods deemed reachable by the algorithm used to compute analysis scope.
Default value: `true`

`chord.std.scope.exclude`

Type: string list
Description: Partial list of prefixes of names of classes, typically inside the JDK standard library, whose methods must be treated as no-ops.
Default value: `""`

`chord.ext.scope.exclude`

Type: string list
Description: Partial list of prefixes of names of classes, typically outside the JDK standard library, whose methods must be treated as no-ops.
Default value: `""`

`chord.scope.exclude`

Type: string list
Description: Complete list of prefixes of names of classes whose methods must be treated as no-ops.
Default value: `"[chord.std.scope.exclude],[chord.ext.scope.exclude]"`

`chord.std.check.exclude`

Type: string list
Description: Partial list of prefixes of names of classes, typically inside the JDK standard library, to be excluded by analyses. Interpretation of this property is analysis-specific.
Default value: `"java.,javax.,sun.,com.sun.,com.ibm., org.apache.harmony."`

`chord.ext.check.exclude`

Type: string list
Description: Partial list of prefixes of names of classes, typically outside the JDK standard library, to be excluded by analyses. Interpretation of this property is analysis-specific.
Default value: `""`

`chord.check.exclude`

Type: string list
Description: Complete list of prefixes of names of classes to be excluded by analyses. Interpretation of this property is analysis-specific.
Default value: `"[chord.std.check.exclude],[chord.ext.check.exclude]"`

5.2.3 Functionality Properties

This section describes properties that dictate what task(s) Chord must perform.

`chord.build.scope`

Type: bool
Description: Compute the analysis scope of the input Java program using the algorithm.
Default value: `false`
Note: The analysis scope is computed regardless of the value of this property if another task (e.g., an analysis specified via property `chord.run.analyses`) demands it.

`chord.run.analyses`

Type: string list
Description: List of names of analyses to be run in order.
Default value: `""`
Note: If the analysis is written in Java, its name is specified via statement `name="..."` in its `@Chord` annotation. If the analysis is written in Datalog, its name is specified via a line of the form "`# name=...`".

`chord.print.methods`

Type: string list
Description: List of methods whose intermediate representation to print to standard output.
Default value: `""`
Note: Specify each method in format `mname:mdesc@cname` where `mname` is the method's name, `mdesc` is the method's descriptor, and `cname` is the name of the method's declaring class. In `cname`, use '.' instead of '/', and use '`#`' instead of '`$`'.

`chord.print.classes`

Type: string list
Description: List of classes whose intermediate representation to print to standard output.
Default value: `""`
Note: In class names, use '.' instead of '/', and use `#` instead of the dollar character.

`chord.print.all.classes`

Type: bool
Description: Print intermediate representation of all classes in scope to standard output.
Default value: `false`

`chord.print.rels`

Type: string list
Description: List of names of program relations whose contents must be printed to files `[chord.out.dir]/<RELNAME>.txt` where `<RELNAME>` denotes the relation name.
Default value: `""`
Note: This functionality must be used with caution as certain program relations, albeit represented compactly as BDDs, may contain a large number (e.g., millions) of tuples, resulting in voluminous output when printed in explicit form to a text file. See Section 15.2 for a more efficient way to query the contents of program relations (namely, by using the `debug` target provided in file `build.xml` in Chord's main directory).

`chord.print.project`

Type: bool
Description: Create files `targets_sortby_name.html`, `targets_sortby_kind.html`, and `targets_sortby_producers.html` in directory `[chord.out.dir]`, publishing all tasks and targets defined by analyses in paths `[chord.java.analysis.path]` and `[chord.dlog.analysis.path]`.
Default value: `false`

`chord.print.results`

Type: bool
Description: Print the results of analyses in HTML. Interpretation of this property is analysis-specific.
Default value: `true`

`chord.verbose`

Type: int in the range [0..5]
Description: Control the verbosity of messages during Chord's execution.
Default value: `1`

5.2.4 Project Properties

This section describes properties regarding analyses executed by Chord.

`chord.classic`

Type: bool
Description: Whether to use the classic project (as opposed to the modern project). See Chapter11 for

the difference between the two kinds of projects.
Default value: `true`

`chord.std.java.analysis.path`

Type: path
Description: Partial classpath of analyses written in Java (i.e., `@Chord`-annotated classes). Conventionally, it includes all Java analyses that are predefined in Chord.
Default value: The absolute path of file `chord.jar`.

`chord.ext.java.analysis.path`

Type: path
Description: Partial classpath of analyses written in Java (i.e., `@Chord`-annotated classes). Conventionally, it includes all user-defined analyses.
Default value: `""`

`chord.java.analysis.path`

Type: path
Description: Complete classpath of analyses written in Java (i.e., `@Chord`-annotated classes).
Default value: `[chord.std.java.analysis.path]:[chord.ext.java.analysis.path]`

`chord.std.dlog.analysis.path`

Type: path
Description: Partial path of analyses written in Datalog (i.e., files with suffix `.dlog`). Conventionally, it includes all Datalog analyses that are predefined in Chord.
Default value: The absolute path of file `chord.jar`.

`chord.ext.dlog.analysis.path`

Type: path
Description: Partial path of analyses written in Datalog (i.e., files with suffix `.dlog`). Conventionally, it includes all user-defined Datalog analyses.
Default value: `""`

`chord.dlog.analysis.path`

Type: path
Description: Complete path of analyses written in Datalog (i.e., files with suffix `.dlog`).
Default value: `[chord.std.dlog.analysis.path]:[chord.ext.dlog.analysis.path]`

5.2.5 Instrumentation Properties

This section describes properties regarding bytecode instrumentation and dynamic analysis.

`chord.use.jvmti`

Type: bool
Description: Whether the JVMTI-based bytecode instrumentation agent from `main/agent/` must be used for running dynamic analyses.
Default value: `false`

`chord.instr.kind`

Type: `[offline|online]`
Description: The kind of bytecode instrumentation. The choices are offline and online (load-time).
Default value: `offline`

`chord.trace.kind`

Type: `[full|pipe]`
Description: The medium by which an event-generating JVM and an event-handling JVM communicate in a dynamic analysis. The choices are regular file and POSIX pipe.
Default value: `full`

`chord.trace.block.size`

Type: int
Description: Number of bytes to read/write in a single operation from/to the event trace file in a multi-JVM dynamic analysis.
Default value: `4096`

`chord.dynamic.haltonerr`

Type: bool
Description: Whether to terminate Chord if the input Java program terminates abnormally during dynamic analysis.
Default value: true

`chord.dynamic.timeout`

Type: int
Description: The amount of time, in milliseconds, after which to kill the process running the given program during dynamic analysis, or -1 if the process must never be killed.
Default value: `-1`

`chord.max.cons.size`

Type: int
Description: Maximum number of bytes over which events generated during the execution of any constructor in the given program may span.
Default value: `50000000`
Note: This property is relevant only for dynamic analyses which want events of the form `BEF_NEW` h t o to be generated (see Section 14.3). The problem with generating such events at run-time is that the ID o of the object freshly created by thread t at object allocation site h cannot be instrumented until the object is fully initialized (i.e., its constructor has finished executing). Hence, Chord first generates a "crude dynamic trace", which has events of the form `BEF_NEW` h t and `AFT_NEW` h t o generated before and after the execution of the constructor, respectively. A subsequent pass generates a "final dynamic trace", which replaces each `BEF_NEW` h t event by `BEF_NEW` h t o. For this purpose, however, Chord must buffer all events generated between the `BEF_NEW` and `AFT_NEW` events, and this property specifies the number of bytes over which these events may span. If the actual number of bytes exceeds the value specified by this property (e.g., if the constructor throws an exception and the `AFT_NEW` event is not generated at all), then Chord simply generates event `BEF_NEW` h i 0 (i.e., it treats the created object as having ID 0, which is the ID also used for `null`).

5.2.6 Caching Properties

This section describes properties that specify what must be reused by Chord, if available, from previous runs instead of recomputing.

`chord.reuse.scope`

Type: bool
Description: Compute analysis scope using the information in files specified by properties `chord.methods.file` and `chord.reflect.file`, if both of those files exist.
Default value: `false`
Note: Property `chord.scope.kind` is ignored if this property is set to `true` and the two files exist.

`chord.reuse.rels`

Type: bool
Description: Load each desired program relation named `<name>` from the BDD stored in file `[chord.bddbddb.work.dir]/<name>.bdd`, if the file exists.
Default value: `false`

`chord.reuse.traces`

Type: bool
Description: Reuse event traces stored in file(s) `chord.trace.file]_full_ver0_runM.txt` for dynamic analysis, if those files exist, where `M` ranges over run IDs specified by property `chord.run.ids`. Property `chord.trace.kind` must be set to `full` if this property is set to `true`.
Default value: `false`

5.2.7 Chord JVM Properties

This section describes properties regarding the JVM that runs Chord.

`chord.max.heap`

Type: string
Description: Maximum heap memory size of the JVM running Chord.
Default value: `1024m`

`chord.max.stack`

Type: string
Description: Maximum thread stack size of the JVM running Chord.
Default value: `32m`

`chord.jvmargs`

Type: string
Description: Arguments to the JVM running Chord.
Default value: `"-showversion -ea -Xmx[chord.max.heap] -Xss[chord.max.stack]"`

5.2.8 BDD Properties

This section describes properties concerning BDD-based Datalog solver bddbddb that is used by Chord to run analyses written in Datalog.

`chord.use.buddy`

Type: bool
Description: Whether BDD library BuDDy from `main/bdd/` must be used by bddbddb.
Default value: `false`

`chord.bddbddb.max.heap`

Type: string
Description: Maximum heap memory size of JVM running bddbddb.
Default value: `1024m`
Note: bddbddb is invoked in a separate JVM for each analysis written in Datalog that is executed. This is primarily because multiple Datalog analyses may be executed in a single run of Chord, resulting in multiple invocations of bddbddb, and it is difficult to reset the state of bddbddb on each invocation.

5.2.9 Output Location Properties

This section describes properties specifying the names of files and directories output by Chord. Most users will not need to alter the default values of these properties.

`chord.out.file`

Type: location
Description: Absolute location of the file to which the standard output stream is redirected during Chord's execution.
Default value: `null`

`chord.err.file`

Type: location
Description: Absolute location of the file to which the standard error stream is redirected during Chord's execution.
Default value: `null`

`chord.out.dir`

Type: location
Description: Absolute location of the directory to which Chord dumps all files.
Default value: `[chord.work.dir]/chord_output/`

`chord.reflect.file`

Type: location
Description: Absolute location of the file from/to which resolved reflection information is read/written.
Default value: `[chord.out.dir]/reflect.txt`

`chord.methods.file`

Type: location
Description: Absolute location of the file from/to which list of methods deemed reachable is read/written.
Default value: `[chord.out.dir]/methods.txt`

`chord.classes.file`

Type: location
Description: Absolute location of the file from/to which list of classes deemed reachable is read/written.
Default value: `[chord.out.dir]/classes.txt`

`chord.bddbddb.work.dir`

Type: location
Description: Absolute location of the directory used by BDD-based Datalog solver bddbddb as its input/output directory (namely, for program domain files `*.dom` and `*.map`, and program relation files `*.bdd`).
Default value: `[chord.out.dir]/bddbddb/`

`chord.boot.classes.dir`

Type: location
Description: Absolute location of the directory from/to which instrumented classes of the input Java program inside the JDK standard library are read/written by dynamic analyses.
Default value: `[chord.out.dir]/boot_classes/`

`chord.user.classes.dir`

Type: location
Description: Absolute location of the directory from/to which instrumented classes of the input Java program outside the JDK standard library are read/written by dynamic analyses.
Default value: `[chord.out.dir]/user_classes/`

`chord.instr.scheme.file`

Type: location
Description: Absolute location of the file specifying the kind and format of events in trace files used by dynamic analyses.
Default value: `[chord.out.dir]/scheme.ser`

`chord.trace.file`

Type: location
Description: Absolute location of trace files used by dynamic analyses.
Default value: `[chord.out.dir]/trace`
Note: Suffix `_full_verN.txt` or `_pipe_verN.txt` is appended to the name of the file, depending upon whether it is a regular file or a POSIX pipe, respectively, where `N` is the version of the file (multiple versions are maintained if the trace is transformed by filters defined by the dynamic analysis; 0 is the final version). If `chord.reuse.traces` is set to `true`, then `_full_verN_runM.txt` is appended to the name of the file, where `M` is the run ID.

Chapter 6

Setting up a Java Program for Analysis

This chapter describes how to setup a Java program for analysis using Chord. Suppose the program has the following directory structure:

```
example/
    src/
        foo/
            Main.java
            ...
    classes/
        foo/
            Main.class
            ...
    lib/
        src/
            taz/
                ...
        jar/
            taz.jar
    chord.properties
```

The above structure is typical: the program's Java source files are under `src/`, its class files are under `classes/`, and the source and jar files of the libraries used by the program are under `lib/src/` and `lib/jar/`, respectively. The purpose of the `chord.properties` file is explained below.

The only way to specify inputs to Chord, including the program to be analyzed, is via system properties. Section 5.1 describes various ways by which properties can be passed to Chord. Here, we describe the simplest approach, in which all properties of the program to be analyzed that might be needed by Chord are defined in a file named

`chord.properties` that is located in the top-level directory of the program (directory `example/` above). Then, Chord can be applied to the program by running the following command:

```
ant -Dchord.work.dir=<WORK_DIR> run
```

This command instructs Chord to run in the directory denoted by `<WORK_DIR>`, where it searches for a file named `chord.properties` and loads all properties defined in that file, if it exists. Thus, for the above program, `<WORK_DIR>` must be the absolute or relative path of the `example/` directory. A sample `chord.properties` file for the above program is as follows:

```
chord.main.class=foo.Main
chord.class.path=classes:lib/jar/taz.jar
chord.src.path=src:lib/src
chord.run.ids=0,1
chord.args.0="-thread 1 -n 10"
chord.args.1="-thread 2 -n 50"
```

Each relative file/directory name in the value of any property defined in this file (e.g., the `lib/src` directory name in the value of property `chord.src.path` above) is treated relative to the directory specified by property `chord.work.dir`, whose default value is the current directory. Section 5.2.1 presents all program properties that are recognized by Chord. Here, we only describe those that are most commonly used, namely, those defined in the above sample properties file:

- `chord.main.class` specifies the fully-qualified name of the main class of the program.
- `chord.class.path` specifies the application classpath of the program (the JDK standard library classpath is implicitly included).
- `chord.src.path` specifies the Java source path of the program. All analyses in Chord operate on Java bytecode. The only use of this property is to HTMLize the Java source files of the program so that the results of analyses can be reported at the Java source code level.
- `chord.run.ids` specifies a list of IDs to identify runs of the program. It is used by dynamic analyses to determine how many times the program must be run. An additional use of this property is to allow specifying the command-line arguments to use in the run having ID `<id>` via property `chord.args.<id>`, as illustrated by properties `chord.args.0` and `chord.args.1` above.

The above command does not do much beyond making Chord load the above properties file. For Chord to do something interesting, additional properties must be set that specify the function(s) Chord must perform. All functions are summarized in Section 5.2.3. The most common function is to run one or more analyses on the input program; it is described in Chapter 13.

Chapter 7

Analysis Scope Construction

A pre-requisite to analyzing a Java program using any program analysis framework, including Chord, is to compute the *analysis scope*: which parts of the program to analyze. Several scope construction algorithms (so-called call-graph algorithms) exist in the literature that differ in scalability (i.e., how large a program they can handle with the available resources) and precision (i.e., how much of the program they deem is reachable).

Chord implements several standard scope construction algorithms. Besides scalability and precision, an additional metric of these algorithms in Chord that can be controlled by users is usability, which concerns aspects such as excluding certain code from being analyzed even if it is reachable, and modeling Java features such as reflection, dynamic class loading, and native methods. These features affect which code are reachable but, in general, they cannot be modeled soundly by any program analysis framework. The best a framework can do is provide stubs for commonly-used native methods in the standard JDK library (e.g., the `arraycopy` method of class `java.lang.System`), offer users a range of options on how to resolve reflection (e.g., an option might be running the program and observing how reflection is resolved), etc.

Chord computes the analysis scope of the given program either if property `chord.build.scope` is set to `true` or if some other task (e.g., a program analysis specified via property `chord.run.analyses`) demands it. The following sections describe Chord's analysis scope computation in detail. Section 7.1 describes how to reuse the analysis scope computed in a previous run of Chord for a given program. Section 7.2 describes Chord's analysis scope construction algorithms. Finally, Section 7.3 describes how users can exclude certain classes from the analysis scope.

7.1 Scope Reuse

If property `chord.reuse.scope` has value `true` and both files specified by properties `chord.methods.file` and `chord.reflect.file` exist, then Chord regards those files as specifying which methods to consider reachable and how to resolve reflection, respectively.

The format of the file specified by property `chord.methods.file` is a list of zero or more lines, where each line is of the form `mname:mdesc@cname` specifying the method's name `mname`, the method's descriptor `mdesc`, and the method's declaring class `cname` (e.g., `main:([Ljava/lang/String;)V@foo.bar.Main`).

The format of the file specified by property `chord.reflect.file` is of the form:

```
# resolvedClsForNameSites
...
# resolvedObjNewInstSites
...
# resolvedConNewInstSites
...
# resolvedAryNewInstSites
...
```

where each of the above "`...`" is a list of zero or more lines, where each line is of the form `bci!mname:mdesc@cname->type1,type2,...typeN` meaning the call site at bytecode offset `bci` in the method denoted by `mname:mdesc@cname` may resolve to any of reference types `type1`, `type2`, ..., `typeN`. The meaning of the above four sections is as follows.

- `resolvedClsForNameSites` lists each call to static method `forName(String)` defined in class `java.lang.Class`, along with a list of the types of the named classes.
- `resolvedObjNewInstSites` lists each call to instance method `newInstance()` defined in class `java.lang.Class`, along with a list of the types of the instantiated classes.
- `resolvedConNewInstSites` lists each call to instance method `newInstance(Object[])` defined in class `java.lang.reflect.Constructor`, along with a list of the types of the instantiated classes.
- `resolvedAryNewInstSites` lists each call to instance method `newInstance(Class,int)` defined in class `java.lang.reflect.Array`, along with a list of the types of the instantiated classes.

The default value of property `chord.reuse.scope` is `false`. The default value of properties `chord.methods.file` and `chord.reflect.file` is `[chord.out.dir]/methods.txt` and `[chord.out.dir]/reflect.txt`, respectively. Property `chord.out.dir` denotes the output directory of Chord; its default value is `[chord.work.dir]/chord_output/`. Property `chord.work.dir` denotes the working directory during Chord's execution; its default value is the current directory.

7.2 Scope Construction Algorithms

If property `chord.reuse.scope` has value `false` or the files specified by properties `chord.methods.file` or `chord.reflect.file` do not exist, then Chord computes analysis scope using the algorithm specified by property `chord.scope.kind` and then writes the list of methods deemed reachable and the reflection resolved by that algorithm to the files specified by properties `chord.methods.file` and `chord.reflect.file`, respectively.

The possible values of property `chord.scope.kind` are [`rta`|`cha`|`dynamic`] (the default value is `rta`). The following subsections describe the scope construction algorithm that Chord runs in each of these three cases. In each case,

Chord at least expects properties `chord.main.class` and `chord.class.path` to be set to the fully-qualified name of the program's main class (e.g., `com.example.Main`) and the program's application classpath, respectively.

7.2.1 Rapid Type Analysis

If property `chord.scope.kind` has value `rta`, then Chord computes analysis scope statically using Rapid Type Analysis (RTA). RTA is an iterative fixed-point algorithm. It maintains a set of reachable methods M. The initial iteration starts by assuming that only the main method in the main class is reachable (Chord also handles class initializer methods but we ignore them here for brevity; we also ignore the set of reachable classes maintained besides the set of reachable methods). All object allocation sites H contained in methods in M are deemed reachable (i.e., control-flow within method bodies is ignored). Whenever a dynamically-dispatching method call site (i.e., an invokevirtual or invokeinterface site) with receiver of static type t is encountered in a method in M, only subtypes of t whose objects are allocated at some site in H are considered to determine the possible target methods, and each such target method is added to M. The process terminates when no more methods can be added.

7.2.2 Class Hierarchy Analysis

If property `chord.scope.kind` has value `cha`, then Chord computes analysis scope statically using Class Hierarchy Analysis (CHA). The key difference between CHA and RTA is that for invokevirtual and invokeinterface sites with receiver of static type t, CHA considers *all* subtypes of t in the class hierarchy to determine the possible target methods, whereas RTA restricts them to types of objects allocated in methods deemed reachable so far. As a result, CHA is highly imprecise in practice, and also expensive since it grossly overestimates the set of reachable classes and methods. Nevertheless, Chord allows users to control which classes are included in the class hierarchy, and thereby control the precision and cost of CHA, by setting property `chord.ch.kind`, whose possible values are [`static`|`dynamic`] (the default value is `static`).

Chord first constructs the entire classpath of the given program by concatenating in order the following classpaths:

1. The boot classpath, specified by property `sun.boot.class.path`.
2. The library extensions classpath, comprising all jar files in directory `[java.home]/lib/dir/`.
3. The application classpath of the given program, specified by property `chord.class.path`, which is empty by default.

All classes in the entire classpath (resulting from items 1–3 above) are included in the class hierarchy with the following exceptions:

- Duplicate classes, i.e., classes with the same name occurring in more than one classpath element; in this case, all occurrences except the first are excluded.
- Any class whose name's prefix is specified in the value of property `chord.scope.exclude` (see Section 7.3).
- If property `chord.ch.kind` has value `dynamic`, then Chord runs the given program and observes the set of all classes the JVM loads; any class not in this set is excluded.

- If the superclass of a class C is missing or if an interface implemented/extended by a class/interface C is missing, where "missing" means that it is either not in the classpath resulting from items 1–3 above or it is excluded by one of these rules, then C itself is excluded. Note that this rule is recursive, e.g., if C has superclass B which in turn has superclass A, and A is missing, then both B and C are excluded.

7.2.3 Dynamic Analysis

If property `chord.scope.kind` has value `dynamic`, then Chord computes analysis scope dynamically, by running the program and observing the classes that are loaded at run-time. The number of times the program is run and the command-line arguments to be supplied to the program in each run is specified by properties `chord.run.ids` and `chord.args.<id>` for each run ID `<id>`. By default, the program is run only once, using run ID `0`, and without any command-line arguments. Only classes loaded in some run are regarded as reachable but *all* methods of each loaded class are regarded as reachable regardless of whether they were invoked in the run. The rationale behind this decision is to both reduce the run-time instrumentation overhead and increase the predictive power of program analyses performed using the computed analysis scope.

7.3 Scope Exclusion

Chord can be instructed to exclude certain classes in a given program from being analyzed. This functionality might be desirable, for instance, if the given program contains a larger framework (e.g., Hadoop or Android) which must not be analyzed. Chord provides three properties for this purpose. The value of each of these properties is a comma-separated list of prefixes of names of classes. Chord treats the body of each method defined in each such class as a no-op.

- Property `chord.std.scope.exclude` is intended to specify classes to be excluded from the scope of *all* programs to be analyzed, e.g., classes in the JDK standard library. Its default value is the empty list.
- Property `chord.ext.scope.exclude` is intended to specify classes to be excluded from the scope of specific programs to be analyzed. Its default value is the empty list.
- Property `chord.scope.exclude` specifies the final list of classes to be excluded from scope. Its default value is `[chord.std.scope.exclude]:[chord.ext.scope.exclude]`.

Note: The value of each of the above properties is a list of *prefixes*, not *regular expressions*. A valid value is "`java.,com.sun.`", but not "`java.*,com.sun.*`".

Chapter 8

Predefined Analyses

Chord provides many standard analyses. This chapter first explains how to run any such analysis and then provides descriptions of various predefined analyses.

8.1 Running an Analysis

Each predefined analysis in Chord has a unique name that can be used to run the analysis from the command-line. The following command runs the analysis named `<ANALYSIS_NAME>` on the program specified by directory `<WORK_DIR>` (see Chapter 6 for how to setup a program):

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=<ANALYSIS_NAME> run
```

For instance, the following command runs a basic may-alias and call-graph analysis (called 0CFA) provided in Chord:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=cipa-0cfa-dlog run
```

This instructs Chord to run the analysis named `cipa-0cfa-dlog`, which is defined in file `main/src/chord/analyses/alias/cipa_0cfa.dlog`.

The output of the above command is of the form:

```
Buildfile: build.xml

run:
     [java] Chord run initiated at: Mar 13, 2011 10:31:08 PM
     [java] ENTER: cipa-0cfa-dlog
     [java] ENTER: T
     [java] ENTER: RTA
     [java] Iteration: 0
     [java] Iteration: 1
     [java] Iteration: 2
     [java] LEAVE: RTA
     [java] SAVING dom T size: 1386
     [java] LEAVE: T
     [java] ENTER: F
     [java] SAVING dom F size: 4120
     [java] LEAVE: F
     ...
     [java] ENTER: MputStatFldInst
     [java] SAVING rel MputStatFldInst size: 739
     [java] LEAVE: MputStatFldInst
     [java] ENTER: statIM
     [java] SAVING rel statIM size: 3359
     [java] LEAVE: statIM
     [java] Starting command: 'java ... chord_analyses_alias_cipa_0cfa.dlog'
     [java] Relation VH: 541 nodes, 449.0 elements (V0,H0)
     [java] Relation FH: 137 nodes, 8.0 elements (H0,F0)
     [java] Relation HFH: 199 nodes, 35.0 elements (H0,F0,H1)
     [java] Relation IM: 590 nodes, 69.0 elements (I0,M0)
     [java] Finished command: 'java ... chord_analyses_alias_cipa_0cfa.dlog'
     [java] LEAVE: cipa-0cfa-dlog
     [java] Chord run completed at: Mar 13, 2011 10:31:36 PM
     [java] Total time: 00:00:27:671 hh:mm:ss:ms

BUILD SUCCESSFUL
Total time: 28 seconds
```

Each analysis in Chord is written modularly, independent of other analyses, along with lightweight annotations specifying the inputs and outputs of the analysis. Chord's runtime automatically computes producer-consumer relationships between analyses (e.g., determines which analysis produces as output a result that is needed as input by another analysis). Before running a desired analysis (such as 0CFA in the above example), Chord recursively runs other analyses until the inputs to the desired analysis have been computed; it finally runs the desired analysis to produce the outputs of that analysis.

The 0CFA analysis consumes and produces multiple *program relations*. The consumed program relations include `MputStatFldInst` and `statIM`, each of which is produced by a separate imperative analysis with the corresponding name, and the produced program relations include `VH`, `FH`, `HFH`, and `IM`. We next briefly discuss these relations.

The program relations consumed by the 0CFA analysis contain basic program facts. For instance, `MputStatFldInst` is a relation containing each tuple (m,f,v) such that method m in the input Java program contains a putstatic instruction of the form "$f = v$", while `statIM` is a relation containing each tuple (i,m) such that m is the target method of invokestatic instruction i.

The program relations produced by the 0CFA analysis represent points-to information and the call graph of the input Java program as computed by the analysis. Specifically, relations `VH`, `FH`, and `HFH` represent points-to information for local variables, static fields, and instance fields, respectively, while relation `IM` represents the call graph, namely, it contain each tuple (i,m) such that m is a possible target method of call site i.

Metavariables m, f, i, and v above range over entities in so-called *program domains* `M`, `F`, `I`, and `V`, respectively. A program domain is a set of entities of a certain kind in the input Java program. For instance, `M` is the domain representing the set of all methods in the input Java program, `F` is the domain representing the set of all fields, `I` is the domain representing the set of all call sites in methods in `M`, and `V` is the domain representing the set of all local variables of reference type in methods in `M`. Each of these domains is produced by a separate Java analysis with the corresponding name. Notice that the analyses producing these domains run upfront because these domains are consumed by the analyses that produce relations such as `MputStatFldInst` and `statIM`, which in turn are consumed by the desired 0CFA analysis.

During execution, Chord dumps intermediate and final results to files in the directory specified by property `chord.out.dir`, whose default value is `[chord.work.dir]/chord_output/` and typically does not need to be changed by users. For the above example, this directory is `<WORK_DIR>/chord_output/`.

The verbosity of Chord's output above is controlled by property `chord.verbose`, whose default value is 1. At verbosity level 0, the above command produces less voluminous output of the form:

```
Buildfile: build.xml

run:
     [java] Chord run initiated at: Mar 13, 2011 10:35:01 PM
     [java] Chord run completed at: Mar 13, 2011 10:35:28 PM
     [java] Total time: 00:00:26:297 hh:mm:ss:ms

BUILD SUCCESSFUL
Total time: 26 seconds
```

8.2 Points-to and Call-Graph Analyses

Chord offers several choices for computing points-to and call-graph information of Java programs. In each of these choices, the points-to and call-graph information is computed simultaneously (called "on-the-fly call-graph construction" in the literature in contrast to "ahead-of-time call-graph construction" in which the call graph is computed first followed by points-to information). On-the-fly approaches are more precise because, in a dynamically dispatching language like Java, as more points-to facts are discovered, more (dynamically dispatched) methods are deemed reachable, thereby growing the call graph; the code in these newly added methods in turn results in more points-to

facts.

Flow-insensitive analysis computes a single abstract heap whereas flow-sensitive analysis computes per-program-point abstract heaps. Context-insensitive analysis analyzes each method at most once (i.e. in a single abstract context), whereas context-sensitive analyses potentially analyze each method multiple times, in different abstract contexts. Thus, flow- and context-sensitive analyses are more precise but less scalable than flow- and context-insensitive analyses, respectively.

Flow-sensitive analysis does not offer much precision over flow-insensitive analysis in practice, especially in the absence of strong updates and in the presence of SSA (Static Single Assignment form), a program representation that renders a flow-insensitive analysis almost as precise as a flow-sensitive analysis. Since analyses in Chord currently perform only weak updates, and since they all operate on an SSA form of the input Java program by default, the rest of this section focuses only on flow-insensitive analysis, which is the predominant kind of points-to/call-graph analysis in Chord.

We describe context-insensitive analysis first because understanding the concepts behind it will help understand the more sophisticated context-sensitive analyses. We first recall some relevant program domains:

- M is the domain of all methods.
- I is the domain of all method call sites.
- F is the domain of all (instance and static) fields.
- V is the domain of all local variables of reference type.
- H is the domain of all object allocation sites.

8.2.1 Context-Insensitive Analysis

The context-insensitive points-to/call-graph analysis treats each object allocation site as a separate abstract memory location; in other words, it can distinguish objects created at different sites but not those created at the same site. Additionally, it is field-sensitive, in that it distinguishes between different instance fields of the same object, but array-insensitive, in that it cannot distinguish between different elements of the same array; all array elements are modeled using a distinguished hypothetical instance field (that has index 0 in domain F).

To run this analysis, run the following command:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=cipa-0cfa-dlog run
```

where `<WORK_DIR>` is a directory containing a file named `chord.properties` that defines properties `chord.main.class` and `chord.class.path` specifying the main class and the application classpath, respectively, of the program to be analyzed.

This analysis outputs the following relations:

- *Call-graph information:*
  - rootM subset M contains the set of entry methods that may be reachable; this includes the program's main method as well as each static initializer method that may be reachable from the main method.
  - reachableM subset M contains the set of methods that may be reachable from the program's main method.
  - IM subset (I × M) contains tuples `(i,m)` such that call site `i` may call method `m`.
- *Points-to information:*
  - FH subset (F × H) contains tuples `(f,h)` such that static field (i.e. global variable) `f` may point to an object allocated at site `h`.
  - VH subset (V × H) contains tuples `(v,h)` such that local variable `v` may point to an object allocated at site `h`.
  - HFH subset (H × F × H) contains tuples `(h1,f,h2)` such that instance field `f` of some object allocated at site `h1` may point to some object allocated at site `h2`.

8.2.2 Context-Sensitive Analysis

In a context-sensitive analysis, there is no longer just one abstract memory location per object allocation site. Rather, the set of objects a reference can point to depends on the *context* in which the method containing the reference is called. Whereas a context-insensitive analysis talks about the domain of methods (M) and the domain of allocation sites (H), a context-sensitive analysis talks about the domain of abstract contexts, labeled C. Elements of domain C contain both abstract calling contexts and abstract objects. (These are merged for reasons described below.)

Chord has several context-sensitive analyses, but they all expose the same relations, which are described below:

- *Context information:*
  - C is the domain of all abstract calling contexts and abstract objects. Each element in this domain is a sequence of zero or more sites, where each site can be a call site or an object allocation site. A sequence may have mixed call sites and object allocation sites. The most significant site (i.e. the first site) in each sequence is called the head; the remaining sub-sequence is called the tail. The below three relations relate a sequence with its head and tail.
  - CC subset (C × C) contains tuples `(c1,c2)` such that context `c2` is the tail of context `c1`.
  - CH subset (C × H) contains tuples `(c,h)` such that object allocation site `h` is the head of context `c`.
  - CI subset (C × I) contains tuples `(c,i)` such that call site `i` is the head of context `c`.
- *Call-graph information:*
  - rootCM subset (C × M) contains tuples `(c,m)` such that method `m` is an entry method in context `c`.
  - reachableCM subset (C × M) contains tuples `(c,m)` such that method `m` may be reachable in context `c`.
  - CICM subset (C × I × C × M) contains tuples `(c1,i,c2,m)` such that call site `i` in context `c1` may call method `m2` in context `c2`.
- *Points-to information:*
  - FC subset (F × C) contains tuples `(f,o)` such that static field (i.e. global variable) `f` may point to object `o`.
  - CVC subset (C × V × C) contains tuples `(c,v,o)` such that local variable `v` may point to object `o` in context `c` of that variable's declaring method. Note that both `o` and `c` are elements of domain C.
  - CFC subset (C × F × C) contains tuples `(o1,f,o2)` such that instance field `f` of object `o1` may point to object `o2`.

Under Construction: explain object-sensitive vs context-sensitive, plus how to invoke.

8.3 Static Datarace Analysis

To run Chord's static datarace analysis, run the following command:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=datarace-java run
```

where `<WORK_DIR>` is a directory containing a file named `chord.properties` that defines properties `chord.main.class`, `chord.class.path`, and `chord.src.path` specifying the main class, the application classpath, and the Java source path, respectively, of the program to be analyzed.

Directory `<CHORD_MAIN_DIR>/examples/datarace_test/` provides a toy Java program on which one can run the datarace analysis. First run `ant` in that directory (in order to compile the program's `.java` files to `.class` files) and then run the above command with `<WORK_DIR>` replaced by `examples/datarace_test/`. Upon successful completion, the following files should be produced in directory `examples/datarace_test/chord_output/`:

- File `dataraces_by_fld.html`, listing all dataraces grouped by the field on which they occur; all dataraces on the same instance field or the same static field are listed in the same group, and so are all dataraces on array elements.
- File `dataraces_by_obj.html`, listing all dataraces grouped by the abstract object on whose field they occur; dataraces on all static fields are listed in the same group, and so are dataraces on different instance fields of the same abstract object.

8.4 Static Deadlock Analysis

To run Chord's static deadlock analysis, run the following command:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=deadlock-java run
```

where `<WORK_DIR>` is a directory containing a file named `chord.properties` that defines properties `chord.main.class`, `chord.class.path`, and `chord.src.path` specifying the main class, the application classpath, and the Java source path, respectively, of the program to be analyzed.

Directory `<CHORD_MAIN_DIR>/examples/deadlock_test/` provides a toy Java program on which one can run the deadlock analysis. First run `ant` in that directory (in order to compile the program's `.java` files to `.class` files) and then run the above command with `<WORK_DIR>` replaced by `examples/deadlock_test/`. Upon successful completion, the file `deadlocks.html` should be produced in directory `examples/deadlock_test/chord_output/`.

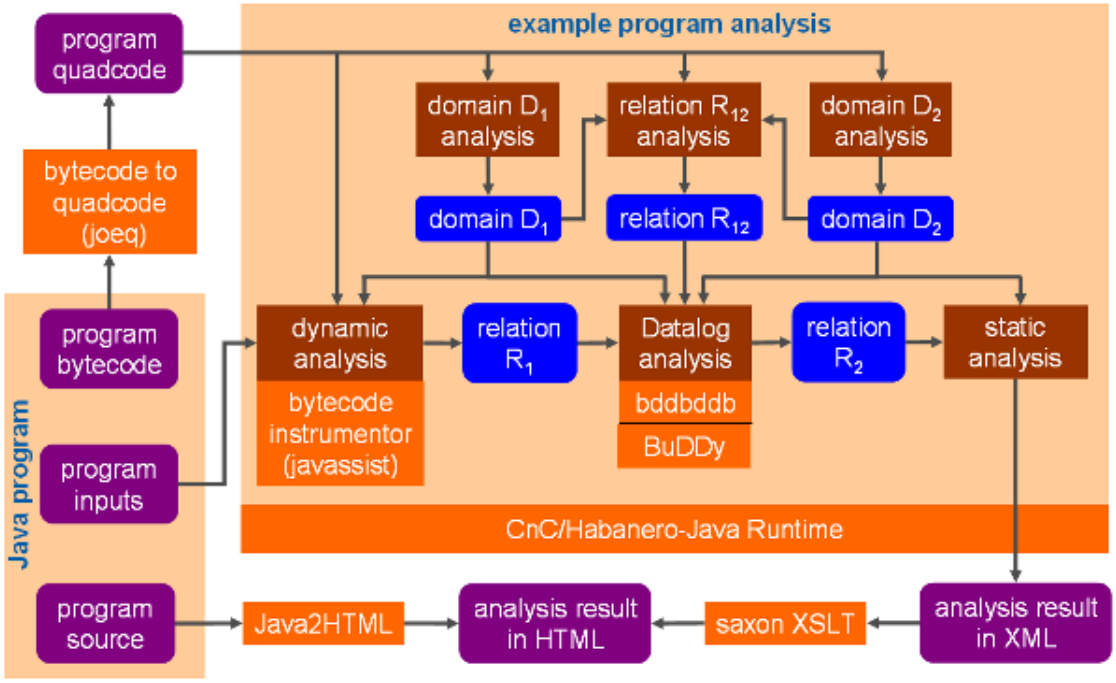
Part III

Guide for Developers

Chapter 9

Architecture of Chord

This chapter presents the high-level architecture of Chord, depicted below, and describes its key components.



Chord Properties

All inputs to Chord are specified by means of system properties. Chapter 5 describes how to set properties and the meaning of each property that is recognized by Chord.

The Java program to be analyzed is also specified via properties. Chapter 6 describes how to setup a Java program for analysis using Chord. Chord analyzes Java bytecode, not Java source code, and thus only requires the program's class files. Certain analyses, however, present their results at the Java source code level, and thus require the program's Java source files as well.

Java Program Representation

Chord uses the Joeq Java compiler framework to convert the Java bytecode of the input Java program, one class file at a time, into a three-address-like intermediate program representation called *quadcode* that is more suitable for analysis. Chapter 10 describes the quadcode representation in detail.

Analysis Scope Construction

A pre-requisite to analyzing a Java program using any program analysis framework, including Chord, is to compute the *analysis scope*: which parts of the program to analyze. Chord implements several standard scope construction algorithms from the literature that differ in aspects such as scalability, precision, and usability for the problem at hand. Chapter 7 describes these algorithms in detail.

Writing and Running Analyses

Chord provides many standard analyses. Chapter 8 describes these analyses and how to run them. Moreover, Chord allows users to define their own analyses, possibly atop the provided analyses.

A distinctive aspect of Chord is that each analysis is written modularly, independent of other analyses, along with lightweight annotations specifying the inputs and outputs of the analysis. Chord's runtime automatically computes dependencies between analyses (e.g., determines which analysis produces as output a result that is needed as input by another analysis). Before running a desired analysis, Chord recursively runs other analyses until the inputs to the desired analysis have been computed; it finally runs the desired analysis to produce the outputs of that analysis.

Chord can be invoked in one of two modes: *classic* or *modern*. These two modes defer in the semantics of dependencies between analyses. In particular, the classic mode is simpler to understand for novice users (the dependencies are only data dependencies) but has a sequential runtime, whereas the modern mode is harder to understand (there are both data and control dependencies) but has a parallel runtime that is capable of running analyses without dependencies between them in parallel. The parallel runtime is based on Habanero-Java, and the semantics of the dependencies between analyses is based on the Habanero Concurrent Collections (CnC) declarative parallel programming model. Chapter 11 expands upon the modular architecture of analyses in Chord.

Chord provides various *analysis templates*: classes containing boilerplate code that can be extended by users to rapidly prototype different kinds of analyses. An example is class RHSAnalysis, named after [Reps, Horowitz, and Sagiv 1995], which can be extended by users to write a summary-based inter-procedural context-sensitive static analysis by merely specifying the abstract domain and intra-procedural transfer functions. Another example is DynamicAnalysis, which can be extended by users to write a dynamic analysis by merely specifying which of various provided events to instrument, and the transfer functions for those events. Chapters 12 and 13 describe how to write and run your own analyses in Chord using the provided analysis templates.

Dynamic Analysis

Chord uses the Javassist Java bytecode manipulation framework for instrumenting bytecode and doing dynamic analysis. Chord offers the most versatile capabilities of any existing dynamic analysis framework for Java, particularly the ability to instrument the entire JDK (including classes in package java.lang). Specifically, it includes support for:

- offline as well as load-time instrumentation of Java bytecode;
- processing of dynamic analysis events online in the same JVM or offline in a different JVM with an uninstrumented JDK (the latter circumvents performance and correctness problems that can arise if a single JVM with an instrumented JDK is used to generate and handle events); and
- allowing the event-generating and event-handling JVMs to run either serially (by storing the entire trace of events to a regular file) or in parallel (by streaming the trace of events in a piped file).

Chapter 14 describes all aspects of dynamic analysis in Chord.

Datalog Analysis

A common way to rapidly prototype an analysis in Chord is using a declarative logic-programming language called Datalog. Chord uses the BDD-based Datalog solver bddbddb to run analyes written in Datalog. Chapter 15 describes all aspects of such analyses.

Chapter 10

Java Program Representation

Chord uses Joeq to translate Java bytecode, one class file at a time, into a three-address-like intermediate representation of the input Java program called *quadcode*. This chapter describes all aspects of quadcode and how it relates to bytecode. It first explains how to pretty-print bytecode and quadcode (Section 10.1) which is useful for debugging analyses and deciphering their output. The remaining sections describe the quadcode representation along with the API of Joeq and Chord for navigating it. Briefly, the representation consists of a set of classes that may be loaded (Section 10.2). The representation of each class consists of a set of members (Section 10.3) which are the fields and methods of the class. The representation of a concrete method (Section 10.4) consists of a control-flow graph (CFG). The representation of a CFG (Section 10.5) consists of a set of registers and a set of basic blocks linked by directed edges denoting flow of control between basic blocks. Each basic block contains zero or more primitive statements called quads (Section 10.6). Finally, the most common way to traverse all quads is discussed (Section 10.7).

10.1 Pretty-Printing

Consider the following Java program contained in file `examples/hello_world/src/test/HelloWorld.java` in Chord's main directory:

```
package test;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

First compile this program by running command `ant` in directory `examples/hello_world/`.

To pretty-print the bytecode representation of a class, run the following command:

```
javap -classpath <CLASS_PATH> -bootclasspath <BOOT_CLASS_PATH> -private -verbose <CLASS_NAME>
```

where:

- `<CLASS_NAME>` is the fully-qualified name of the class whose bytecode is to be printed (`test.HelloWorld` in our example).
- `<CLASS_PATH>` is the classpath of that class (`examples/hello_world/` in our example).
- `<BOOT_CLASS_PATH>` is the boot classpath; it is optional and must be supplied if `<CLASS_NAME>` is a class from the JDK standard library (e.g., `java.util.ArrayList`) that has been modified and written to a user-defined location (e.g., it has been instrumented by Chord and written to `chord_output/boot_classes/`).

Program `javap` comes along with the JVM. The output of the above command for our example is as follows:

```
Compiled from "HelloWorld.java"
public class test.HelloWorld extends java.lang.Object
  SourceFile: "HelloWorld.java"
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Method   #6.#20; //  java/lang/Object."<init>":()V
const #2 = Field    #21.#22;    //  java/lang/System.out:Ljava/io/PrintStream;
const #3 = String   #23;    //  Hello World!
const #4 = Method   #24.#25;    //  java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = class    #26;    //  test/HelloWorld
const #6 = class    #27;    //  java/lang/Object
const #7 = Asciz    <init>;
const #8 = Asciz    ()V;
const #9 = Asciz    Code;
const #10 = Asciz   LineNumberTable;
const #11 = Asciz   LocalVariableTable;
const #12 = Asciz   this;
const #13 = Asciz   Ltest/HelloWorld;;
const #14 = Asciz   main;
const #15 = Asciz   ([Ljava/lang/String;)V;
const #16 = Asciz   args;
const #17 = Asciz   [Ljava/lang/String;;
```

```
const #18 = Asciz   SourceFile;
const #19 = Asciz   HelloWorld.java;
const #20 = NameAndType #7:#8;//  "<init>":()V
const #21 = class   #28;    //  java/lang/System
const #22 = NameAndType #29:#30;//  out:Ljava/io/PrintStream;
const #23 = Asciz   Hello World!;
const #24 = class   #31;    //  java/io/PrintStream
const #25 = NameAndType #32:#33;//  println:(Ljava/lang/String;)V
const #26 = Asciz   test/HelloWorld;
const #27 = Asciz   java/lang/Object;
const #28 = Asciz   java/lang/System;
const #29 = Asciz   out;
const #30 = Asciz   Ljava/io/PrintStream;;
const #31 = Asciz   java/io/PrintStream;
const #32 = Asciz   println;
const #33 = Asciz   (Ljava/lang/String;)V;

{
public test.HelloWorld();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
  LineNumberTable:
   line 3: 0
  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      5      0    this       Ltest/HelloWorld;

public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc #3; //String Hello World!
   5:   invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return
  LineNumberTable:
   line 5: 0
   line 6: 8
  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      9      0    args       [Ljava/lang/String;
}
```

To pretty-print the quadcode representation of a class, run the following command:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.print.classes=<CLASS_NAME> \
    -Dchord.verbose=0 -Dchord.out.file=<OUT_FILE> run
```

where:

- `<WORK_DIR>` is a directory (`examples/hello_world/` in our example) that contains a file named `chord.properties` which defines properties `chord.main.class` and `chord.class.path` specifying the main class and the classpath of the input Java program. Alternatively, these two properties can be defined directly on the above command-line.
- `<CLASS_NAME>` is the fully-qualified name of the class whose quadcode is to be printed (`test.HelloWorld` in our example). Each occurrence of a '`$`' in the class name must be replaced by a '`#`'.
- `<OUT_FILE>` is the file to which the quadcode must be written; if left unspecified, the quadcode is written to the standard output.

The output of the above command for our example is as follows:

```
*** Class: test.HelloWorld
Method: main:([Ljava/lang/String;)V@test.HelloWorld
    0#1
    5#3
    5#2
    8#4
Control flow graph for main:([Ljava/lang/String;)V@test.HelloWorld:
BB0 (ENTRY) (in: <none>, out: BB2)

BB2 (in: BB0 (ENTRY), out: BB1 (EXIT))
1: GETSTATIC_A T1, .out
3: MOVE_A T2, AConst: "Hello World!"
2: INVOKEVIRTUAL_V println:(Ljava/lang/String;)V@java.io.PrintStream, (T1, T2)
4: RETURN_V

BB1 (EXIT)  (in: BB2, out: <none>)

Exception handlers: []
Register factory: Registers: 3
```

10.2 Whole Program

This and the following sections describe the quadcode representation along with the API of Joeq and Chord for navigating it. This API is contained in packages `chord.program`, `joeq.Class`, and `joeq.Compiler.Quad`.

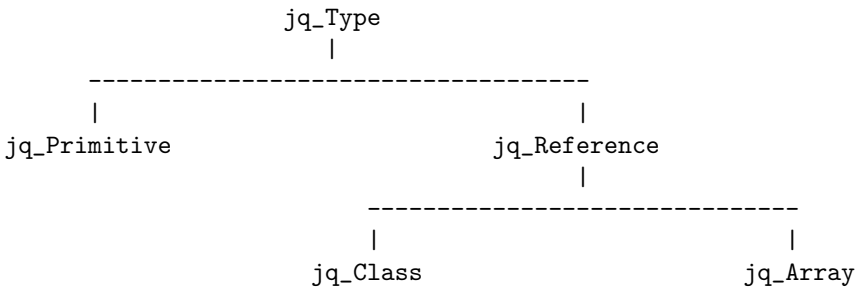
The quadcode representation of the whole program is a unique global object of class `chord.program.Program` which can be obtained by calling static method `chord.program.Program.g()`. This class provides a rich API (in the form of public instance methods) to access various parts of the representation, most notably:

`IndexSet<jq_Type> getTypes()`	All types referenced in classes that may be loaded.
`IndexSet<jq_Reference> getClasses()`	All classes that may be loaded.*
`IndexSet<jq_Method> getMethods()`	All methods that may be called.

* Includes both classes/interfaces and array types, represented as objects of `jq_Class` and `jq_Array`, respectively; both these are subclasses of `jq_Reference`.

See Chapter 7 for how Chord determines which classes may be loaded and which methods may be called.

The quadcode representation of each type is a unique object of the appropriate subclass of `joeq.Class.jq_Type` in the following hierarchy:



10.3 Class Members

Each primitive type (e.g., boolean, int, etc.) is represented by a unique `jq_Primtive` object. Each class and each interface type is represented by a unique `jq_Class` object. Each array type is represented by a unique `jq_Array` object.

Members (i.e., fields and methods) of the class/interface represented by an object of class `joeq.Class.jq_Class` can be accessed using the following API provided by that class.

`String getName()`	Fully-qualified name of the class, e.g., "`java.lang.String[]`".
`jq_InstanceField[] getDeclaredInstanceFields()`	All instance fields declared in the class.
`jq_StaticField[] getDeclaredStaticFields()`	All static fields declared in the class.
`jq_InstanceMethod[] getDeclaredInstanceMethods()`	All instance methods declared in the class.
`jq_StaticMethod[] getDeclaredStaticMethods()`	All static methods declared in the class.

Chord uses format `mName:mDesc@cName`, described in class `chord.program.MethodSign`, to uniquely identify each field and each method in the input Java program, where `mName` denotes the name of the field/method, `mDesc` denotes the descriptor of the field/method (see below), and `cName` denotes the fully-qualified name of the class declaring the field/method. For instance, "`main:[Ljava/lang/String;@test.HelloWorld`" uniquely identifies the main method in the example above. We next review field descriptors and method descriptors from the Java bytecode specification.

A field descriptor represents the type of a local variable or a (static or instance) field. It is a series of characters generated by the grammar:

```
FieldDescriptor : FieldType
      FieldType : BaseType | ObjectType | ArrayType
       BaseType : B | C | D | F | I | J | S | Z
     ObjectType : L <classname> ;
      ArrayType : [ ComponentType
  ComponentType : FieldType
```

The characters of `BaseType`, the '`L`' and '`;`' of `ObjectType`, and the '`[`' of `ArrayType` are all ASCII characters. The `<classname>` represents a fully qualified class or interface name. The interpretation of the field types is as shown in the below table:

BaseType Character	Type	Interpretation
`B`	byte	signed byte
`C`	char	Unicode character
`D`	double	double-precision floating-point value
`F`	float	single-precision floating-point value
`I`	int	integer
`J`	long	long integer
`L<classname>;`	reference	an instance of class `<classname>`
`S`	short	signed short
`Z`	boolean	true or false
`[`	reference	one array dimension

For example, the descriptor of type `int` is simply `I`. The descriptor of an instance variable of type `Object` is "`Ljava/lang/Object;`". Note that the internal form of the fully qualified name for class `Object` is used. The descriptor of a multidimensional double array of type "`double[][][]`" is "`[[[D`".

A method descriptor represents the types of the arguments and return result of a method:

```
   MethodDescriptor : ( ParameterDescriptor* ) ReturnDescriptor
ParameterDescriptor : FieldType
   ReturnDescriptor : FieldType | V
```

A parameter descriptor represents the type of an argument of a method. A return descriptor represents the type of the return result of a method. The character '`V`' indicates that the method returns no value (its return type is void).

The method descriptor is the same whether it is a static or an instance method. Although an instance method is passed `this`, a reference to the current class instance, in addition to its intended arguments, that fact is not reflected in the method descriptor.

For example, the method descriptor for the method "`Objectfoo(inti,doubled,Threadt)`" is "`(IDLjava/lang/Thread;)Ljava/lang/Object;`". Note that internal forms of the fully qualified names of `Thread` and `Object` are used in the method descriptor.

10.4 Methods

The quadcode representation of each method is a unique object of class `joeq.Class.jq_Method`. Components of the method, most notably its control-flow graph, can be accessed using the following API provided by that class.

`String getName()`	Name of the method.
`String getDesc().toString()`	Descriptor of the method, e.g., "`(Ljava/lang/String;)V`".
`jq_Class getDeclaringClass()`	Declaring class of the method.
`ControlFlowGraph getCFG()`	Control-flow graph of the method.*
`int getLineNumber(int bci)`	Line number of the given bytecode offset (-1 if not found).
`Quad getQuad(int bci)`	First quad at the given bytecode offset (null if not found).
`Quad getQuad(int bci, Class kind)`	First quad of the given kind at the given bytecode offset (null if not found).
`Quad getQuad(int bci, Class[] kind)`	First quad of any given kind at the given bytecode offset (null if not found).
`String toString()`	Unique identifier of the method in format `mName:mDesc@cName`.

* The control-flow graph must not be asked if the method is abstract (which can be determined by calling instance method `isAbstract()` of `jq_Method`).

10.5 Control-Flow Graphs

The control-flow graph (CFG) of each method consists of a set of registers, called the register factory, and a directed graph whose nodes are basic blocks and whose edges denote flow of control between basic blocks.

The CFG of each method is a unique object of class `joeq.Compiler.Quad.ControlFlowGraph`. Components of the CFG can be accessed using the following API provided by that class.

`getRegisterFactory()`	Set of all local variables.
`EntryOrExitBasicBlock entry()`	Unique entry basic block.
`EntryOrExitBasicBlock exit()`	Unique exit basic block.
`ListIterator.BasicBlock reversePostOrderIterator()`	Iterator over all basic blocks in reverse post-order.
`jq_Method getMethod()`	Containing method of the CFG.

The register factory contains one register per argument of the method (called *local variables*) and one register per temporary in the method body (called *stack variables*). Temporaries include those declared by programmers as well as those generated by Joeq. The reason Joeq can generate temporaries is that the quadcode representation, which is register-based, is constructed from Java bytecode, which is stack-based; moreover, Joeq does the Static Single Assignment (SSA) transformation by default, which introduces temporaries to ensure that there is at most one static assignment to any variable. Registers corresponding to local variables are named `R0`, `R1`, ..., `Rn`, while those

corresponding to stack variables are named `Tn+1`, `Tn+2`, ..., `Tm`.

For instance, the register factory of the main method in the example above has 3 registers: `R0` denoting the `args` argument of the method and `T1` and `T2` denoting temporaries generated by Joeq.

Each register factory is a unique object of class `joeq.Compiler.Quad.RegisterFactory`.

Besides the register factory, a CFG has a directed graph whose nodes are basic blocks and whose edges denote flow of control between basic blocks. Each basic block contains a straight-line sequence of zero or more primitive statements called quads (Section 10.6). Each CFG is guaranteed to contain at least two basic blocks: a unique entry basic block with no incoming edges and a unique exit block with no outgoing edges. The entry and exit basic blocks do not contain any quads.

Each basic block is a unique object of class `joeq.Compiler.Quad.BasicBlock` (the entry and exit basic blocks are instances of a subclass `joeq.Compiler.Quad.EntryOrExitBasicBlock`). Components of the basic block can be accessed using the following API provided by that class.

`int size()`	Number of quads contained in the basic block.
`Quad getQuad(int index)`	Quad at the given 0-based index.
`List.BasicBlock getPredecessors()`	List of immediate predecessor basic blocks.
`List.BasicBlock getSuccessors()`	List of immediate successor basic blocks.
`jq_Method getMethod()`	Containing method of the basic block.

10.6 Quads

Chord uses format `offset!mName:mDesc@cName`, described in class `chord.program.MethodElem`, to uniquely identify each bytecode instruction in the input Java program, where `offset` is the (0-based) bytecode offset of the instruction in its containing method, `mName` is the name of the method, `mDesc` is the descriptor of the method, and `cName` is the fully-qualified name of the class declaring the method. For instance, "`8!main:[Ljava/lang/String;@test.HelloWorld`" uniquely identifies the return instruction in the main method in the example above.

The quadcode representation is register-based, as opposed to Java bytecode that is used to construct it, which is stack-based. As a result, it uses *quads* to represent bytecode instructions. A quad is a primitive statement that consists of an operator and upto four operands. There is no one-to-one correspondence between bytecode instructions and quads: certain bytecode instructions generate a sequence of more than one quads while others do not generate any quad. The API of class `jq_Method` provides various `getQuad(...)` methods to access the quad(s) corresponding to a bytecode instruction (see Section 10.4).

Each quad is a unique object of class `joeq.Compiler.Quad.Quad`. Components of the quad can be accessed using the following API provided by that class.

`Operator getOperator()`	Kind of the quad.
`int getBCI()`	Bytecode offset of the quad in its containing method.
`String toByteLocStr()`	Unique identifier of the quad in format `offset!mName:mDesc@cName`.
`String toJavaLocStr()`	Location of the quad in format `fileName:lineNum` in Java source code.
`String toLocStr()`	Location of the quad in both Java bytecode and source code.
`String toVerboseStr()`	Verbose description of the quad (its location plus contents).
`jq_Method getMethod()`	Containing method of the quad.

The kind of each quad is determined by its operator which is a unique object of the appropriate subclass of `joeq.Compiler.Quad.Operator` in the following hierarchy:

```
Operator
    |
    |--- Move
    |--- Phi
    |--- Unary
    |--- Binary
    |--- New
    |--- NewArray
    |--- MultiNewArray
    |--- Getstatic
    |--- Putstatic
    |--- ALoad
    |--- AStore
    |--- Getfield
    |--- Putfield
    |--- CheckCast
    |--- InstanceOf
    |--- ALength
    |--- Return
    |
    |--- Branch
    |       |
    |       |--- IntIfCmp
    |       |--- Goto
    |       |--- Jsr
    |       |--- Ret
    |       |--- LookupSwitch
    |       |--- TableSwitch
    |
    |--- Invoke
    |       |
    |       |--- InvokeVirtual
    |       |--- InvokeStatic
    |       |--- InvokeInterface
    |
    |--- Monitor
            |
```

```
        |--- MONITORENTER
        |--- MONITOREXIT
```

The number and kinds of operands of each quad depends upon the kind of the operator. Each of the above subclasses of `Operator` provides an API to access the operands of the quad. For instance, the components of a `Getfield` quad `q` of the form "l = b.f" can be accessed as follows:

```
Operand lo = Getfield.getSrc(q);
Operand bo = Getfield.getBase(q);
if (lo instanceof RegisterOperand && bo instanceof RegisterOperand) {
    Register l = ((RegisterOperand) lo).getRegister();
    Register b = ((RegisterOperand) bo).getRegister();
    jq_Field f = Getfield.getField(q).getField();
    ...
}
```

10.7 Traversing Quadcode

A common way to traverse all quads in the input Java program is as follows:

```
import chord.program.Program;
import joeq.Compiler.Quad.QuadVisitor;
import joeq.Class.jq_Method;
import joeq.Compiler.Quad.ControlFlowGraph;
import joeq.Util.Templates.ListIterator;
import joeq.Compiler.Quad.BasicBlock;
import joeq.Compiler.Quad.Quad;

QuadVisitor qv = new QuadVisitor.EmptyVisitor() {
    public void visitMove(Quad q) { ... }
    public void visitPhi(Quad q) { ... }
    public void visitUnary(Quad q) { ... }
    ...
};
Program program = Program.g();
for (jq_Method m : program.getMethods()) {
    if (!m.isAbstract()) {
        ControlFlowGraph cfg = m.getCFG();
        ListIterator.BasicBlock it = cfg.reversePostOrderIterator();
        while (it.hasNext()) {
            BasicBlock b = it.nextBasicBlock();
            for (int i = 0; i < b.size(); i++) {
                Quad q = b.getQuad(i);
                q.accept(qv);
            }
        }
    }
}
```

Chapter 11

A Chord Project: Tasks, Targets, and Dependencies

In order to facilitate heavy reuse and rapid prototyping, each analysis in Chord is written modularly, independent of other analyses, along with lightweight annotations specifying the inputs and outputs of the analysis. In each run, upon startup, Chord organizes all analyses and their inputs and outputs (collectively called analysis results) using a global entity called a *project*. More concretely, a project consists of a set of analyses called *tasks*, a set of analysis results called *targets*, and a set of data/control dependencies between tasks and targets.

The project built in a particular run is of either of the following two kinds, depending upon whether the value of property `chord.classic` is true or false, respectively.

- a *classic project*, represented as an object of class `chord.project.ClassicProject`.
- a *modern project*, represented as an object of class `chord.project.ModernProject`.

The project representation can be obtained by calling static method `g()` of the corresponding class. A classic project is built by default. The two kinds of projects differ primarily in that the only kind of dependencies in a classic project are data dependencies whereas both data and control dependencies are allowed in a modern project. The key advantage of a modern project is that it can schedule independent tasks in parallel whereas a classic project always runs tasks sequentially. This chapter focusses on classic projects as the runtime for modern projects is still under development. We next explain how Chord builds a classic project (a set of tasks, a set of targets, and a set of dependencies between them).

Tasks: There are two kinds of tasks corresponding to the two broad kinds of analyses in Chord. They are summarized in the following table:

Kind:	imperative (see Chapter 12)	declarative (see Chapter 15)
Location:	a `.class` file in the path denoted by property `chord.java.analysis.path` compiled from a `@Chord`-annotated class implementing interface `chord.project.ITask`	a `.dlog` file in the path denoted by property `chord.dlog.analysis.path`
Name:	via stmt `name="<NAME>"` in `@Chord` annotation	via line "`# name=<NAME>`" in `.dlog` file
Form:	an instance of the `@Chord`-annotated class	an instance of class `chord.project.analyses.DlogAnalysis`

Each task in Chord is of the form "`\{C1,...,Cn\}T\{P1,...,Pm\}`" where:

- `T` is the code provided by the user to be executed when the task is executed,
- `C1`, ..., `Cn` are the names of zero or more targets specified by the user as being consumed by the task, and
- `P1`, ..., `Pm` are the names of zero or more targets specified by the user as being produced by the task.

The consumed targets may be produced by other tasks and, likewise, the produced targets may be consumed by other tasks.

Targets: The set of targets in a project includes each target that is specified as consumed/produced by some task in the project. When defining tasks, the user implicitly or explicitly provides the class (type) of each target. Chord reports a runtime error if a target has no type or has multiple types. Otherwise, it creates a separate instance of that class to represent that target.

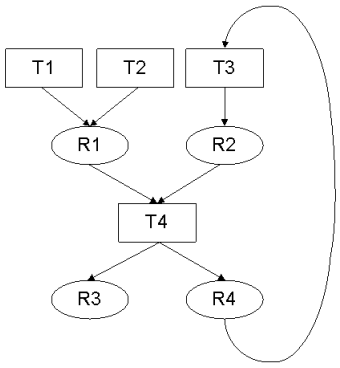
Dependencies: Chord computes a dependency graph as a directed graph whose nodes are all tasks and targets computed as above, and:

- There is an edge from a target C to a task T if the user has specified that T consumes C.
- There is an edge from a task T to a target P is the user has specified that T produces P.

We next present an example project to illustrate various concepts in the rest of this chapter:

```
{} T1 { R1 }
{} T2 { R2 }
{ R4} T3 { R2 }
{ R1, R2 } T4 { R3, R4 }
```

The set of tasks in this project is { `T1`, `T2`, `T3`, `T4` } and the set of targets in the project is { `R1`, `R2`, `R3`, `R4` }. The dependency graph is as follows:



Class `chord.project.ClassicProject` provides a rich API (in the form of public instance methods) for accessing tasks and targets in the project, for running tasks, and for resetting tasks and targets. The most commonly used methods are as follows:

`ITask getTask(String name)`	Representation of the task named `name`.
`Object getTrgt(String name)`	Representation of the target named `name`.
`ITask runTask(String name)`	Execute the task named `name`.
`boolean isTaskDone(String name)`	Whether task named `name` has alread been executed.
`boolean isTrgtDone(String name)`	Whether target named `name` has already been computed.
`void setTaskDone(String name)`	Force task named `name` to not be executed the next time it is demanded.
`void setTrgtDone(String name)`	Force target named `name` to not be computed the next time it is demanded.
`void resetTaskDone(String name)`	Force task named `name` to be executed the next time it is demanded.
`void resetTrgtDone(String name)`	Force target named `name` to be computed the next time it is demanded.

We next explain the above methods.

The `getTask(name)` provides the representation of the unique task with the specified name, if it exists, and a runtime error otherwise.

The `getTrgt(name)` provides the representation of the unique target with the specified name, if it exists, and a runtime error otherwise.

A "done" bit, initialized to false, is kept with each task and each target in the project. The `runTask(name)` method runs the task with the specified name, if it exists, and reports a runtime error otherwise. Running a task proceeds as follows. If the done bit of the task is true, no action is taken. Otherwise, suppose the task is of the form "`\{C1,...,Cn\}T\{P1,...,Pm\}`". Then, the following two actions are taken in order:

1. For each of the consumed targets `C1`, ..., `Cn` whose done bit is false, the unique task in the project producing that target is run recursively. A runtime error is reported if no such task exists or if multiple such tasks exist.
2. Once all consumed targets are done, the code `T` of this task itself is run.
3. Finally, the done bit of this task as well as of each of its produced tasks `P1`, ..., `Pn` is set to true.

It is the user's responsibility to ensure termination in the case in which there are cycles in the dependency graph. The `isTaskDone(name)` and `isTrgtDone(name)` methods can be used in the code `T` of any task to enquire whether

the "done" bit of the task or target with the specified name is set to true. Moreover, methods `setTaskDone(name)`, `setTrgtDone(name)` can be used to set the done bit of the task or target with the specified name to true, and likewise, methods `resetTaskDone(name)` and `resetTrgtDone(name)` can be used to set the done bit to false.

It is possible to run tasks from the command-line of Chord by specifying the value of property `chord.run.analyses` as a comma-separated list of the names of tasks to be run in order (see Chapter 13).

We next illustrate the above concepts using the above example. Suppose Chord is run with the value of property `chord.run.analyses` as "`T4`". This causes `runTask(T4)` to be called. The done bit of task `T4` is initialized to false. Hence, the done bit of its first consumed target `R1` is checked. Since it is also initialized to false, the unique task producing target `R1` is demanded. However, multiple tasks `T1` and `T2` producing target `R1` are found in the project, resulting in a runtime error which reports the ambiguity between tasks `T1` and `T2`.

To resolve the ambiguity (say in favor of task `T1`), the user can specify the value of property `chord.run.analyses` as "`T1`,`T4`". This time, `runTask(T1)` is called followed by `runTask(T4)`. Since the done bit of task `T1` is initialized to false and it has no consumed targets, `runTask(T1)` simply executes the code of task `T1`, and sets the done bit of task `T1` and of its only produced target `R1` to true. Next, the call to `runTask(T4)` proceeds as described in the previous run above, but this time the done bit of target `R1` consumed by task `T4` is set to true. Hence, the demand for the unique task producing target `R1` (and the ensuing ambiguity runtime error) is averted. However, this time a different problem occurs: the done bit of the other target `R2` consumed by task `T4` is initialized to false, which results in a call to `runTask(T3)` (since task `T3` is the unique task that produces target `R2`), which in turn results in a call to `runTask(T4)`. The result is infinite mutually-recursive calls to `runTask(T4)` and `runTask(T3)` unless the code of task `T3` or `T4` averts it by calling `setTaskDone` or `setTrgtDone` on some task or target in the cycle. This scenario resulting from a cycle in the dependence graph is rare in practice. It typically occurs in the case of iterative refinement-based client-driven analyses: the output of such an analysis in one iteration is fed as an input to the same analysis in a subsequent iteration. The code of such an analysis (task) must explicitly control execution as described above to avert infinite recursion.

Chapter 12

Writing an Analysis

Chord provides several *analysis templates*: classes containing boilerplate code that can be extended by users to rapidly prototype different kinds of analyses. These classes are organized in the following hierarchy in package `chord.project.analyses`:

```
JavaAnalysis
    |
    |--- ProgramDom
    |
    |--- ProgramRel
    |
    |--- DlogAnalysis
    |
    |--- RHSAnalysis
    |         |
    |         |--- ForwardRHSAnalysis
    |         |
    |         |--- BackwardRHSAnalysis
    |
    |--- BasicDynamicAnalysis
              |
              |--- DynamicAnalysis
```

The following sections describe each of these analysis templates in more detail.

12.1 JavaAnalysis

Class `chord.project.analyses.JavaAnalysis` is the most general template for writing an analysis. An analysis can be created using this template by extending this class as follows:

```
import chord.project.Chord;
import chord.project.ClassicProject;
import chord.project.analyses.JavaAnalysis;
import chord.program.Program;

@Chord(
    name = "<ANALYSIS_NAME>",
    consumes = { "C1", ..., "Cn" },
    produces = { "P1", ..., "Pm" },
    namesOfTypes = { "C1", ..., "Cn", "P1", ..., "Pm" },
    types = { A1.class, ..., An.class, B1.class, ..., Bm.class }
)
public class ExampleAnalysis extends JavaAnalysis {
    @Override public void run() {
        Program program = Program.g();
        ClassicProject project = ClassicProject.g();
        A1 c1 = (A1) project.getTrgt("C1");
        ...
        An cn = (An) project.getTrgt("Cn");
        B1 p1 = (B1) project.getTrgt("P1");
        ...
        Bm pm = (Bm) project.getTrgt("Pm");
        // compute produced targets p1, ..., pm from program and
        // consumed targets c1, ..., cn
        ...
    }
}
```

To run the analysis, class `ExampleAnalysis` must be compiled to a `.class` file that occurs in some element (directory or jar/zip file) of the path specified by property `chord.java.analysis.path`. This causes the analysis to be included in a Chord project as a task that is represented as a separate object of class `ExampleAnalysis`.

The `@Chord` annotation, defined in class `chord.project.Chord`, specifies via fields the following aspects of the analysis:

- Field `name` specifies the name of the analysis (`<ANALYSIS_NAME>`).
- Field `consumes` specifies the names of targets that are consumed by the analysis (`C1`, ..., `Cn`).
- Field `produces` specifies the names of targets that are produced by the analysis (`P1`, ..., `Pm`).
- Fields `namesOfTypes` and `types` specify the types of targets. There is a 1-to-1 correspondence between the arrays denoted by these two fields, e.g., the type of the target named `C1` is class `A1`, and so on. In principle, the type of *any* target in the project can be specified here. In practice, however, the types of only the targets

declared as consumed/produced by this analysis are specified. Moreover, although the type of each target that is consumed/produced by the above analysis is specified in the above annotation, in practice the types of hardly any targets need to be explicitly specified, because they can be automatically inferred by Chord from analyses created using more specialized templates discussed below that also consume/produce those targets.

The code of the analysis must be supplied in the `run()` method. This method typically does the following in order: (1) retrieves the program being analyzed and the representation of each consumed/produced target from the project; (2) performs some computation that uses the program and the consumed targets as inputs; and (3) writes the outputs of the computation to the produced targets.

The analysis templates presented in the following sections are more specialized forms of the `JavaAnalysis` template: they constrain the number and kinds of consumed/produced targets and/or the analysis code in the `run()` method.

12.2 ProgramDom

Class `chord.project.analyses.ProgramDom` is a template for writing a *program domain analysis*. A *program domain* represents an indexed set of values of a fixed kind, typically from the program being analyzed, such as the set of all methods in the program, the set of all fields in the program, etc. Indices are assigned starting from 0 and in the order in which values are added to the set. A program domain primarily serves as an input to Datalog analyses (see Section 12.4). Thus, it is a kind of target (i.e., analysis result) in a Chord project. A common way to define a program domain is to create a program domain analysis by extending class `ProgramDom` as follows:

```
import chord.project.Chord;
import chord.project.ClassicProject;
import chord.project.analyses.ProgramDom;
import chord.program.Program;

@Chord(
    name = "<DOM_NAME>",
    consumes = { "C1", ..., "Cn" }
)
public class ExampleDom extends ProgramDom<DOM_TYPE> {
    @Override public void fill() {
        Program p = Program.g();
        ClassicProject project = ClassicProject.g();
        A1 c1 = (A1) project.getTrgt("C1");
        ...
        An cn = (An) project.getTrgt("Cn");
        // populate domain using program and consumed targets c1, ..., cn
        for (...) {
            DOM_TYPE e = ...;
            add(e);
        }
```

```
    }
}
```

To run the analysis, class `ExampleDom` must be compiled to a `.class` file that occurs in some element (directory or jar/zip file) of the path specified by property `chord.java.analysis.path`. This causes the analysis to be included in a Chord project as a task that is represented as a separate object of class `ExampleDom`. Moreover, that object also denotes a target in the Chord project. Both the task and target have the same name `<DOM_NAME>`.

The `ProgramDom` template can be viewed as providing the following specialized form of the general `JavaAnalysis` template:

- It consumes any number and kinds of targets explicitly declared in the `@Chord` annotation (`C1`, ..., `Cn`).
- It produces a single target, namely, the defined program domain itself (named `<DOM_NAME>` and of type `ExampleDom`). It is a runtime error to explicitly declare any produced targets in the `@Chord` annotation (see below).
- Its `run()` method adds values to the defined program domain. Typically, it suffices to override the `fill()` method (which is called by the `run()` method) and call from it the `add(e)` method for each value `e` to be added to the domain in order.

It is a runtime error to explicitly specify any produced targets in the `@Chord` annotation of a class extending `ProgramDom`. If you wish to define an analysis that produces additional targets besides a program domain, then you can still define the program domain in a class such as `ExampleDom` that extends `ProgramDom`, but you must not annotate it with the `@Chord` annotation (since this annotation causes the class to be egarded as defining an analysis). Instead, define the analysis in a separate class that extends `JavaAnalysis`, as follows:

```
import chord.project.Chord;
import chord.project.ClassicProject;
import chord.project.analyses.JavaAnalysis;

@Chord(
    name = "<ANALYSIS_NAME>",
    consumes = { "C1", ..., "Cn" },
    produces = { "<DOM_NAME>", ... }
)
public class ExampleAnalysis extends JavaAnalysis {
    @Override public void run() {
        ExampleDom d = (ExampleDom) ClassicProject.g().getTrgt("<DOM_NAME>");
        d.run();  // produce domain named <DOM_NAME>
        ...
    }
}
```

Note that targets `C1`, ..., `Cn` that were declared as consumed in the `@Chord` annotation of class `ExampleDom` (and any other fields such as `namesOfTypes` and `types`) must now be provided in the `@Chord` annotation of class `ExampleAnalysis`.

12.3 ProgramRel

Class `chord.project.analyses.ProgramRel` is a template for writing a *program relation analysis*. A *program relation* represents a set of tuples over one or more fixed program domains. A program relation primarily serves as an input or output of Datalog analyses (see Section 12.4). Thus, it is a kind of target (i.e., analysis result) in a Chord project. A common way to define a program relation is to create a program domain analysis by extending class `ProgramRel` as follows:

```
import chord.project.Chord;
import chord.project.ClassicProject;
import chord.project.analyses.ProgramDom;
import chord.project.analyses.ProgramRel;
import chord.program.Program;

@Chord(
    name = "<REL_NAME>",
    consumes = { "C1", ..., "Cn" },
    sign = "<DOM_NAMES>:<DOM_ORDER>"
)
public class ExampleRel extends ProgramRel {
    @Override public void fill() {
        Program p = Program.g();
        ProgramDom<T1> d1 = doms[0];
        ...
        ProgramDom<Tm> dm = doms[m-1];
        ClassicProject project = ClassicProject.g();
        A1 c1 = (A1) project.getTrgt("C1");
        ...
        An cn = (An) project.getTrgt("Cn");
        // populate relation using program, its domains d1, ..., dm, and
        // consumed targets c1, ..., cn
        for (...) {
            T1 o1 = ...;
            Tm om = ...;
            add(o1, ..., om);
        }
    }
}
```

To run the analysis, class `ExampleRel` must be compiled to a `.class` file that occurs in some element (directory or jar/zip file) of the path specified by property `chord.java.analysis.path`. This causes the analysis to be included in a Chord project as a task that is represented as a separate object of class `ExampleRel`. Moreover, that object also denotes a target in the Chord project. Both the task and target have the same name `<REL_NAME>`.

The `ProgramRel` template can be viewed as providing the following specialized form of the general `JavaAnalysis` template:

- It consumes any number and kinds of targets explicitly declared in the `@Chord` annotation (`C1`, ..., `Cn`) as well as implicit targets corresponding to the program domains in the schema of the defined program relation (named `D1`, ..., `Dm` and having type `ProgramDom`).
- It produces a single target, namely, the defined program relation itself (named `<REL_NAME>` and of type `ExampleRel`). It is a runtime error to explicitly declare any produced targets in the `@Chord` annotation (see below).
- Its `run()` method adds tuples to the defined program relation. Typically, it suffices to override the `fill()` method (which is called by the `run()` method) and call from it the `add(o1, ..., om)` method for each tuple (`o1`, ..., `om`) to be added to the relation.

Unlike for program domains, the order in which tuples are added to a program relation is irrelevant. But the relative ordering of the program domains over which the program relation is declared matters heavily for performance. This is because each program relation in Chord is represented symbolically (as oppoosed to explicitly) using a data structure called a Binary Decision Diagram (BDD for short). This in turn is because in pratice, a program relation (e.g., one representing context-sensitive points-to information) can contain millions or billions of tuples even for a moderately-sized input Java program; representing such a large number of tuples explicitly is prohibitively and needlessly expensive. The size of a BDD, on the other hand, does not depend at all upon the number of tuples in the program relation that the BDD represents. Instead, it depends heavily upon the relative ordering of the program domains over which the program relation is declared. Hence, the `@Chord` annotation on a class such as `ExampleRel` that extends `ProgramRel` is required to have a `sign` field whose value is the *sign* of the program relation. A sign is a string of the form `<DOM_NAMES>:<DOM_ORDER>` where:

- `<DOM_NAMES>` is mandatory and specifies the relation's *schema*: a comma-separated list of names of the domains over which the relation is defined, with each domain name suffixed with a non-negative integer that is typically 0 and must be unique across multiple occurrences of the same domain name. The order of domain names in the schema is *irrelevant*: users must pick this order purely based on what order they find most convenient to remember.

  For example, suppose the program relation represents the result of Class Hierarchy Analysis (CHA), i.e., it contains each tuple of the form (m_1,t,m_2) such that method m_2 is the resolved method of an invokevirtual or invokeinterface call with resolved method m_1 on an object of class t. Let `M` and `T` denote the names of the program domains representing the set of all methods and the set of all classes, respectively, in the Java program being analyzed. Then, `<DOM_NAMES>` for this program relation could be any of "`M0,T0,M1`", "`M1,T0,M0`", "`M0,T1,M1`", and so on.
- `<DOM_ORDER>` is optional and determines the relation's representation as a BDD. It is a permutation of the names in `DOM_NAMES>` with a separator '_' or 'x' between consecutive names. The order of names in this list, and the kind of separators used between them, are what determines both the size of the BDD and the performance of operations on it (such as join, selection, projection, etc.).

An example value of `<DOM_ORDER>` for the CHA program relation above with `<DOM_LIST>` as "`M0,T0,M1`" is "`M0xM1_T0`"; another example value is "`M1_T0_M0`".

Chapter 15 describes how BDDs are represented (Section 15.3) and how you can tune their size and the performance of operations on them (Section 15.2).

It is a runtime error to explicitly specify any produced targets in the `@Chord` annotation of a class extending `ProgramRel`. If you wish to define an analysis that produces additional targets besides a program relation, then you can still define the program relation in a class such as `ExampleRel` that extends `ProgramRel`, but you must not annotate it with the `@Chord` annotation (since this annotation causes the class to be egarded as defining an analysis). Instead, define the analysis in a separate class that extends `JavaAnalysis`, as follows:

```
import chord.project.Chord;
import chord.project.ClassicProject;
import chord.project.analyses.JavaAnalysis;

@Chord(
    name = "<ANALYSIS_NAME>",
    consumes = { "C1", ..., "Cn" },
    produces = { "<REL_NAME>", ... },
    namesOfSigns = { "<REL_NAME>", ... },
    signs = { "<DOM_ORDER>:<DOM_NAME>", ... }
)
public class ExampleAnalysis extends JavaAnalysis {
    @Override public void run() {
        ExampleRel r = (ExampleRel) ClassicProject.g().getTrgt("<REL_NAME>");
        r.run();  // produce program relation named <REL_NAME>
    }
}
```

Note that targets `C1`, ..., `Cn` that were declared as consumed in the `@Chord` annotation of class `ExampleRel` (and any other fields such as `namesOfTypes` and `types`) must now be provided in the `@Chord` annotation of class `ExampleAnalysis`. Just like the 1-to-1 correspondence between the values of fields `namesOfTypes` and `types`, there is a 1-to-1 correspondence between the values of fields `namesOfSigns` and `signs`, which allow relating the name of any program relation in the project with its sign.

12.4 DlogAnalysis

A common way to rapidly prototype analyses in Chord is using a declarative logic-programming language called Datalog. A Datalog analysis is defined in a `.dlog` file that primarily specifies the following:

- A set of *program domains* `D1`, ..., `Dk`. A program domain is a set of values of a fixed kind. Each program domain in a Chord project is a target that is represented as a separate object of class `ProgramDom` (see Section 12.2).
- A set of *program relations*, including input relations `I1`, ..., `In` and output relations `O1`, ..., `Om`. A program relation is a set of tuples over one or more fixed program domains `D1`, ..., `Dk`. Each program relation in a Chord project is a target that is represented as a separate object of class `ProgramRel` (see Section 12.3).
- A set of rules (constraints) `R` specifying how to compute the output relations from the input relations.

An example such file is shown in Chapter 15 which also explains all aspects of Datalog analyses.

To run the analysis, the `.dlog` file must occur in some element (directory or jar/zip file) of the path specified by property `chord.dlog.analysis.path`. This causes the analysis to be included in a Chord project as a task that is represented as a separate object of class `chord.project.analyses.DlogAnalysis`. Note that, unlike class `JavaAnalysis`, users must not extend class `DlogAnalysis` but must instead define the analysis in a `.dlog` file.

The `DlogAnalysis` template can be viewed as providing the following specialized form of the general `JavaAnalysis` template:

- It consumes targets `D1`, ..., `Dk` of type `ProgramDom` and `I1`, ..., `In` of type `ProgramRel`.
- It produces targets `O1`, ..., `Om` of type `ProgramRel`.
- Its `run()` method executes Datalog solver `bddbddb` to compute output relations `O1`, ..., `Om` from input relations `I1`, ..., `In` by solving constraints `R`.

12.5 DynamicAnalysis

Under construction

12.6 RHSAnalysis

Under construction

Chapter 13

Running an Analysis

This chapter describes how to run an analysis in Chord. Broadly, there are two kinds of analyses in Chord, summarized in the following table:

Kind:	imperative (see Chapter 12)	declarative (see Chapter 15)
Location:	a `.class` file in the path denoted by property `chord.java.analysis.path` compiled from a `@Chord`-annotated class implementing interface `chord.project.ITask`	a `.dlog` file in the path denoted by property `chord.dlog.analysis.path`
Name:	via stmt `name="<NAME>"` in `@Chord` annotation	via line "`# name=<NAME>`" in `.dlog` file

In its most general form, the command for running an analysis is as follows:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=<ANALYSIS_NAME> \
    -Dchord.dlog.analysis.path=<DLOG_ANALYSIS_PATH> \
    -Dchord.java.analysis.path=<JAVA_ANALYSIS_PATH> run
```

where:

- `<WORK_DIR>` is a directory containing a file named `chord.properties` that defines various properties of the program to be analyzed that might be needed by the analysis being run, such as the program's main class, the program's application classpath, etc. (see Chapter 6).
- `<ANALYSIS_NAME>` is the name of the analysis to run. More generally, it can be a comma-separated list of names of analyses to run in order.
- `<JAVA_ANALYSIS_PATH>` is a path specifying all imperative analyses that might be needed to run the desired analysis.

- `<DLOG_ANALYSIS_PATH>` is a path specifying all declarative analyses that might be needed to run the desired analysis.

In order to facilitate heavy reuse and rapid prototyping, each analysis in Chord is written modularly, independent of other analyses, along with lightweight annotations specifying the inputs and outputs of the analysis. Chord's runtime automatically computes producer-consumer relationships between analyses (e.g., determines which analysis produces as output a result that is needed as input by another analysis). Before running the desired analysis named `<ANALYSIS_NAME>`, Chord recursively runs other analyses until the inputs to the desired analysis have been computed; it finally runs the desired analysis to produce the outputs of that analysis. Chapter 11 explains this process in detail. Each of these analyses must occur in the path denoted by `<JAVA_ANALYSIS_PATH>` or `<DLOG_ANALYSIS_PATH>`, depending upon whether the analysis is written imperatively or declaratively, respectively.

Chord provides shorthand ways for specifying analysis paths by means of the following six properties:

Analysis Kind	Predefined	User-defined	All
imperative	`chord.std.java.analysis.path`	`chord.ext.java.analysis.path`	`chord.java.analysis.path`
declarative	`chord.std.dlog.analysis.path`	`chord.ext.dlog.analysis.path`	`chord.dlog.analysis.path`

The paths specified by the `chord.std.*.analysis.path` properties conventionally include all "standard" analyses: analyses that are predefined in Chord. The default value of each of these properties is the absolute path of file `chord.jar`. Normally, users must not change the value of these properties.

The paths specified by the `chord.ext.*.analysis.path` properties conventionally include all "external" analyses: analyses that are written by users. The default value of each of these properties is the empty path.

The paths specified by the `chord.*.analysis.path` properties include *all* analyses: both standard and external ones. The default value of each of these properties is simply the concatenation of the values of the corresponding two properties above that specify the paths of standard and external analyses. Normally, users must not change the value of these properties.

Running the above command can cause Chord to report a runtime error in the following scenarios:

- Either no included analysis or multiple included analyses are named `<ANALYSIS_NAME>`.
- A result declared as consumed by the analysis named `<ANALYSIS_NAME>` (or by some analysis on which the specified analysis is dependent directly or transitively) is either not declared as produced by any included analysis or is declared as produced by multiple included analyses.

To fix the error resulting from the "missing analysis" case in both the above scenarios, simply include the missing analysis in the path specified by properties `chord.*.analysis.path`.

To fix the error resulting from the "ambiguous analysis" case in both the above scenarios, the names `A1`, ..., `An` of all analyses that were involved in the ambiguity are provided in the error report. Suppose `Ai` is the desired analysis from this set. Then, one way to fix the error is to exclude all analyses in that set except analysis `Ai` from the path specified by properties `chord.*.analysis.path`. A better way is to specify the names of *multiple* analyses in the value of property `chord.run.analyses` (recall that this property allows a comma-separated list of names of analyses to be run in order). Specifically, the value of this property must specify the name of the desired analysis `Ai` *before* the name of the analysis that caused the ambiguity error.

The above command is for users who have defined their own analyses and wish to run them. The following simpler command that uses the default values for properties `chord.*.analysis.path` suffices for users who only wish to run analyses predefined in Chord:

```
ant -Dchord.work.dir=<WORK_DIR> -Dchord.run.analyses=<ANALYSIS_NAME> run
```

Section 8.1 illustrates this command using an example predefined analysis in Chord.

Chapter 14

Dynamic Analysis

This chapter describes all aspect of dynamic analysis in Chord. Section 14.1 describes how to write a dynamic analysis, Section 14.2 describes how to compile and run it, and Section 14.3 describes common dynamic analysis events supported in Chord.

14.1 Writing a Dynamic Analysis

Follow the following steps to write your own dynamic analysis.

Create a class extending `chord.project.analyses.DynamicAnalysis` and override the appropriate methods in it. The only methods that must be compulsorily overridden are method `getInstrScheme()`, which must return an instance of the "instrumentation scheme" required by your dynamic analysis (i.e., the kind and format of events to be generated during an instrumented program's execution) plus each `process<event>(<args>)` method that corresponds to event `<event>` with format `<args>` enabled by the chosen instrumentation scheme. See Section 14.3 for the kinds of supported events and their formats.

A sample such class `MyDynamicAnalysis` is shown below:

```
import chord.project.Chord;
import chord.project.analyses.DynamicAnalysis;
import chord.instr.InstrScheme;

// ***TODO***: analysis won't be recognized by Chord without this annotation
@Chord(name = "<ANALYSIS_NAME>")
public class MyDynamicAnalysis extends DynamicAnalysis {
    InstrScheme scheme;
    @Override
    public InstrScheme getInstrScheme() {
        if (scheme != null)
            return scheme;
        scheme = new InstrScheme();
        // ***TODO***: Choose (<event1>, <args1>), ... (<eventN>, <argsN>)
        // depending upon the kind and format of events required by this
        // dynamic analysis to be generated for this during an instrumented
        // program's execution.
        scheme.set<event1>(<args1>);
        ...
        scheme.set<eventN>(<argsN>);
        return scheme;
    }
    @Override
    public void initAllPasses() {
        // ***TODO***: User code to be executed once and for all
        // before all instrumented program runs start.
    }
    @Override
    public void doneAllPasses() {
        // ***TODO***: User code to be executed once and for all
        // after all instrumented program runs finish.
    }
    @Override
    public void initPass() {
        // ***TODO***: User code to be executed once before each instrumented program run starts.
    }
    @Override
    public void donePass() {
        // ***TODO***: User code to be executed once after each instrumented program run finishes.
    }
    @Override
    public void process<event1>(<args1>) {
        // ***TODO***: User code for handling events of kind <event1> with format <args1>.
    }
    ...
    @Override
    public void process<eventN>(<argsN>) {
        // ***TODO***: User code for handling events of kind <eventN> with format <argsN>.
    }
```

```
}
```

14.2 Compiling and Running a Dynamic Analysis

Compile the analysis by placing the directory containing class `MyDynamicAnalysis` created above in the path defined by property `chord.ext.java.analysis.path`.

Provide the IDs of program runs to be generated (say 1, 2, ..., M) and the command-line arguments to be used for the program in each of those runs (say `<args1>`, ..., `<argsM>`) via properties `chord.run.ids=1,2,...,N` and `chord.args.1=<args1>`, ..., `chord.args.M=<argsM>`. By default, `chord.run.ids=0` and `chord.args.0=""`, that is, the program will be run only once (using run ID 0) with no command-line arguments.

To run the analysis, set property `chord.run.analyses` to `<ANALYSIS_NAME>` (recall that `<ANALYSIS_NAME>` is the name provided in the `@Chord` annotation for class `MyDynamicAnalysis` created above).

Note: The IBM J9 JVM on Linux is highly recommended if you intend to use Chord for dynamic program analysis, as it allows you to instrument the entire JDK; using any other platform will likely require excluding large parts of the JDK from being instrumented. Additionally, if you intend to use online (load-time) bytecode instrumentation in your dynamic program analysis, then you will need JDK 6 or higher, since this functionality requires the `java.lang.instrument` API with class retransformation support (the latter support is available only in JDK 6 and higher).

You can change the default values of various properties for configuring your dynamic analysis; see Section 5.2.2 and Section 5.2.5 in Chapter 5. For instance:

- You can set property `chord.scope.kind` to `dynamic` so that the program scope is computed dynamically (i.e., by running the program) instead of statically.
- You can exclude certain classes (e.g., JDK classes) from being instrumented by setting properties `chord.std.scope.exclude`, `chord.ext.scope.exclude`, and `chord.scope.exclude`.
- You can choose between online (i.e. load-time) and offline bytecode instrumentation by setting property `chord.instr.kind` to `online` or `offline`.
- You can require the event-generating and event-handling JVMs to be one and the same (by setting property `chord.trace.kind` to `none`), or to be separate (by setting property `chord.trace.kind` to `full` or `pipe`, depending upon whether you want the two JVMs to exchange events by a regular file or a POSIX pipe, respectively). Using a single JVM can cause correctness/performance issues if event-handling Java code itself is instrumented (e.g., say the event-handling code uses class `java.util.ArrayList` which is not excluded from program scope). Using separate JVMs prevents such issues since the event-handling JVM runs uninstrumented bytecode (only the event-generating JVM runs instrumented bytecode). If a regular file is used to exchange events between the two JVMs, then the JVMs run serially: the event-generating JVM first runs to completion, dumps the entire dynamic trace to the regular file, and then the event-handling JVM processes the dynamic trace. If a POSIX pipe is used to exchange events between the two JVMs, then the JVMs run in lockstep. Obviously, a pipe is more efficient for very long traces, but it not portable (e.g., it does not currently work on Windows/Cygwin), and the traces cannot be reused across Chord runs (see the following item).

- You can reuse dynamic traces from a previous Chord run by setting property `chord.reuse.traces` to `true`. In this case, you must also set property `chord.trace.kind` to `full`.

- You can set property `chord.dynamic.haltonerr` to `false` to prevent Chord from terminating even if the program on which dynamic analysis is being performed crashes.

Chord offers much more flexibility in crafting dynamic analyses. You can define your own instrumentor (by subclassing `chord.instr.CoreInstrumentor` instead of using the default `chord.instr.Instrumentor`) and your own event handler (by subclassing `chord.runtime.CoreEventHandler` instead of using the default `chord.runtime.EventHandler`). You can ask the dynamic analysis to use your custom instrumentor and/or your custom event handler by overriding methods `getInstrumentor()` and `getEventHandler()`, respectively, defined in `chord.project.analyses.CoreDynamicAnalysis`. Finally, you can define your own dynamic analysis template by subclassing `chord.project.analyses.CoreDynamicAnalysis` instead of subclassing the default `chord.project.analyses.DynamicAnalysis`.

14.3 Common Dynamic Analysis Events

Chord provides support for instrumenting common dynamic analysis events. The below table describes these events.

Event Kind	**Description**
EnterMainMethod(**t**)	After thread **t** enters method **m** (in domain M).
EnterMethod(**m**, **t**)	After thread **t** enters method **m** (in domain M).
LeaveMethod(**m**, **t**)	Before thread **t** leaves method **m** (in domain M).
EnterLoop(**w**, **t**)	Before thread **t** begins loop **w** (in domain W).
LoopIteration(**w**, **t**)	Before thread **t** starts a new iteration of loop **w** (in domain W).
LeaveLoop(**w**, **t**)	After thread **t** finishes loop **w** (in domain W).
BasicBlock(**b**, **t**)	Before thread **t** enters basic block **b** (in domain B).
Quad(**p**, **t**)	Before thread **t** executes quad at program point **p** (in domain P).
BefMethodCall(**i**, **t**, **o**)	Before thread **t** executes the method invocation statement at program point **i** (in domain I) with this argument as object **o**. **Note:** Not generated before constructor calls; use BefNew event.
AftMethodCall(**i**, **t**, **o**)	After thread **t** executes the method invocation statement at program point **i** (in domain I) with this argument as object **o**. **Note:** Not generated after constructor calls; use AftNew event.
BefNew(**h**, **t**, **o**)	Before thread **t** executes a `new` bytecode instruction and allocates fresh object **o** at program point **h** (in domain H).
AftNew(**h**, **t**, **o**)	After thread **t** executes a `new` bytecode instruction and allocates fresh object **o** at program point **h** (in domain H).
NewArray(**h**, **t**, **o**)	After thread **t** executes a `newarray` bytecode instruction and allocates fresh object **o** at program point **h** (in domain H).
GetstaticPrimitive(**e**, **t**, **f**)	After thread **t** reads primitive-typed static field **f** (in domain F) at program point **e** (in domain E).
GetstaticReference(**e**, **t**, **f**, **o**)	After thread **t** reads object **o** from reference-typed static field **f** (in domain F) at program point **e** (in domain E).
PutstaticPrimitive(**e**, **t**, **f**)	After thread **t** writes primitive-typed static field **f** (in domain F) at program point **e** (in domain E).
PutstaticReference(**e**, **t**, **f**, **o**)	After thread **t** writes object **o** to reference-typed static field **f** (in domain F) at program point **e** (in domain E).
GetfieldPrimitive(**e**, **t**, **b**, **f**)	After thread **t** reads primitive-typed instance field **f** (in domain F) of object **b** at program point **e** (in domain E).
GetfieldReference(**e**, **t**, **b**, **f**, **o**)	After thread **t** reads object **o** from reference-typed instance field **f** (in domain F) of object **b** at program point **e** (in domain E).
PutfieldPrimitive(**e**, **t**, **b**, **f**)	After thread **t** writes primitive-typed instance field **f** (in domain F) of object **b** at program point **e** (in domain E).
PutfieldReference(**e**, **t**, **b**, **f**, **o**)	After thread **t** writes object **o** to reference-typed instance field **f** (in domain F) of object **b** at program point **e** (in domain E).
AloadPrimitive(**e**, **t**, **b**, **i**)	After thread **t** reads the primitive-typed element at index **i** of array object **b** at program point **e** (in domain E).
AloadReference(**e**, **t**, **b**, **i**, **o**)	After thread **t** reads object **o** from the reference-typed element at index **i** of array object **b** at program point **e** (in domain E).
AstorePrimitive(**e**, **t**, **b**, **i**)	After thread **t** writes the primitive-typed element at index **i** of array object **b** at program point **e** (in domain E).
AstoreReference(**e**, **t**, **b**, **i**, **o**)	After thread **t** writes object **o** to the reference-typed element at index **i** of array object **b** at program point **e** (in domain E).
ThreadStart(**i**, **t**, **o**)	Before thread **t** calls the `start()` method of `java.lang.Thread` at program point **i** (in domain I) and spawns a thread **o**.
ThreadJoin(**i**, **t**, **o**)	Before thread **t** calls the `join()` method of `java.lang.Thread` at program point **i** (in domain I) to join with thread **o**.
AcquireLock(**l**, **t**, **o**)	After thread **t** executes a statement of the form "monitorenter **o**" or enters a method synchronized on **o** at program point **l** (in domain L).
ReleaseLock(**r**, **t**, **o**)	Before thread **t** executes a statement of the form "monitorexit **o**" or leaves a method synchronized on **o** at program point **r** (in domain R).
Wait(**i**, **t**, **o**)	Before thread **t** calls the `wait()` method of `java.lang.Object` on object **o** at program point **i** (in domain I).

Chapter 15

Datalog Analysis

A common way to rapidly prototype an analysis in Chord is using a declarative logic-programming language called Datalog. This chapter describes all aspects of Datalog analyses in Chord. Section 15.1 explains how to write and run a Datalog analysis. Section 15.2 explains how to tune its performance. Finally, Section 15.3 explains the representation of BDDs (Binary Decision Diagrams) which are a data structure central to executing Datalog analyses.

15.1 Writing a Datalog Analysis

A Datalog analysis declares a bunch of input/output program relations, each over one or more program domains, and provides a bunch of rules (constraints) specifying how to compute the output relations from the input relations. It can be defined in any file with suffix `.dlog` in any directory in the path specified by property `chord.dlog.analysis.path`. An example Datalog analysis is shown below:

```
# name=datarace-dlog

# Program domains
.include "E.dom"
.include "F.dom"
.include "T.dom"

# BDD variable order
.bddvarorder E0xE1_T0_T1_F0

# Input/intermediate/output program relations
field(e:E0,f:F0) input
write(e:E0) input
reach(t:T0,e:E0) input
alias(e1:E0,e2:E1) input
escape(e:E0) input
unguarded(t1:T0,e1:E0,t2:T1,e2:E1) input
hasWrite(e1:E0,e2:E1)
candidate(e1:E0,e2:E1)
datarace(t1:T0,e1:E0, t2:T1,e2:E1) output

# Analysis constraints
hasWrite(e1,e2) :- write(e1).
hasWrite(e1,e2) :- write(e2).
candidate(e1,e2) :- field(e1,f), field(e2,f), hasWrite(e1,e2), e1 <= e2.
datarace(t1,e1,t2,e2) :- candidate(e1,e2), reach(t1,e1), reach(t2,e2), \
    alias(e1,e2), escape(e1), escape(e2), unguarded(t1,e1,t2,e2).
```

Any line that begins with a "`#`" is regarded a comment, except a line of the form "`#name=<ANALYSIS_NAME>`", which specifies the name `<ANALYSIS_NAME>` of the Datalog analysis. Each Datalog analysis is expected to have exactly one such line. The above Datalog analysis is named `datarace-dlog`.

The "`.include"<DOM_NAME>.dom"`" lines specify each domain named `<DOM_NAME>` that is needed by the Datalog analysis, i.e., each domain over which any relation that is input/output by the Datalog analysis is defined. The declaration of each such relation specifies the relation's schema: all the domains over which the relation is defined. If the same domain appears multiple times in a relation's schema then contiguous integers starting from 0 must be used to distinguish them; for instance, in the above example, `candidate` is a binary relation, both of whose attributes have domain `E`, and they are distinguished as `E0` and `E1`.

Each relation is represented symbolically (as opposed to explicitly) using a graph-based data structure called a Binary Decision Diagram (BDD for short). Each domain containing N elements is assigned log2(N) BDD variables. The size of a BDD and the efficiency of operations on it depends heavily on the order of these BDD variables. The "`.bddvarorder<BDD_VAR_ORDER>`" line in the Datalog analysis enables the Datalog analysis writer to specify this order. It must list all domains along with their numerical suffixes, separated by '`_`' or '`x`'. Using a '`_`' between two domains, such as `T0_T1`, means that the BDD variables assigned to domain `T0` precede those assigned to domain `T1` in the BDD variable order for this Datalog analysis. Using a '`x`' between two domains, such as `E0xE1`, means that

the BDD variables assigned to domains `E0` and `E1` will be interleaved in the BDD variable order for this Datalog analysis. See Section 15.3 for more details on BDD representations.

Each Datalog analysis rule is a Horn clause of the form "`R(t) :- R1(t1), ..., Rn(tn)`" meaning that if relations `R1`, ..., `Rn` contain tuples `t1`, ..., `tn` respectively, then relation `R` contains tuple `t`. A backslash may be used at the end of a line to break long rules for readability. The Datalog analysis solver bddbddb used in Chord does not apply any sophisticated optimizations to simplify the rules; besides the BDD variable order, the manner in which these rules are expressed heavily affects the performance of the solver. For instance, an important manual optimization involves breaking down long rules into multiple shorter rules communicating via intermediate relations. See Section 15.2 for hints on tuning the performance of Datalog analyses.

Running a Datalog analysis is no different from running any other kind of analysis in Chord. See Chapter 13 for how to run an analysis.

15.2 Tuning Performance

There are several tricks analysis writers can try to improve the performance of bddbddb, the Datalog solver used by Chord, often by several orders of magnitude. Try these tricks by running the following command:

```
ant -Ddlog.file=<FILE> -Dwork.dir=<DIR> solve
```

where `<FILE>` is the file defining the Datalog analysis to be tuned, and `<DIR>` is the directory containing the program domains (`*.dom` files) and program relations (`*.bdd` files) consumed by the analysis (this is by default the `chord_output/bddbddb/` directory generated by a previous run of Chord.

1. Set properties `noisy=yes`, `tracesolve=yes`, and `fulltracesolve=yes` on the above command line and observe which rule gets "stuck" (i.e., takes several seconds to solve). `fulltracesolve` is seldom useful, but `noisy` and `tracesolve` are often very useful. Once you identify the rule that is getting stuck, it will also tell you which relations and which domains used in that rule, and which operation on them, is taking a long time to solve. Then try to fix the problem with that rule by doing either or both of the following:
   - Break down the rule into multiple rules by creating intermediate relations (the more relations you have on the RHS of a rule the slower it generally takes to solve that rule).
   - Change the relative order of the domains of those relations in the BDD variable order (note that you can use either '_' or 'x' between a pair of domains).
2. Once you have ensured that none of the rules is getting "stuck", you will notice that some rules are applied too many times, and so although each application of the rule itself isn't taking too much time, the cumulative time for the rule is too much. After finishing solving a Datalog analysis, bddbddb prints how long each rule took to solve (both in terms of the number of times it was applied and the cumulative time it took). It sorts the rules in the order of the cumulative time. You need to focus on the rules that took the most time to solve

(they will be at the bottom of the list). Assuming you removed the problem of rules getting "stuck", the rules will roughly be in the order of the number of times they were applied. Here is an example:

```
OUT> Rule VH(u:V0,h:H0) :- VV(u:V0,v:V1), VH(v:V1,h:H0), VHfilter
(u:V0,h:H0).
OUT>    Updates: 2871
OUT>    Time: 6798 ms
OUT>    Longest Iteration: 0 (0 ms)
OUT> Rule IM(i:I0,m:M0) :- reachableI(i:I0), specIMV(i:I0,m:M0,v:V0), VH(v:V0,_:H0).
OUT>    Updates: 5031
OUT>    Time: 6972 ms
OUT>    Longest Iteration: 0 (0 ms)
```

Notice that the second rule was applied 5031 times whereas the first was applied 2871 times. More importantly, the second rule took 6972 milliseconds in all, compared to 6798 for the first rule. Hence, you should focus on the second rule first, and try to speed it up. This means that you should focus only on relations IM, reachableI, specIMV, and VH, and the domains I0, M0, V0, and H0. Any changes you make that do not affect these relations and domains are unlikely to make your solving faster. In general, look at the last few rules, not just the last one, and try to identify the "sub-analysis" of the Datalog analysis that seems problematic, and then focus on speeding up just that sub- analysis.

3. You can add the `.split` keyword at the end of certain rules as a hint to bddbddb to decompose those rules into simpler ones that can be solved faster. You can also set property `split_all_rules=yes` as shorthand for splitting all rules without adding the `.split` keyword to any of them, though I seldom find splitting all rules helpful.

4. You can try to decompose a single Datalog analysis file into two separate Datalog analysis files. Of course, you cannot separate mutually-recursive rules into two different analyses, but if you unnecessarily club together rules that could have gone into different analyses, then they can put conflicting demands on bddbddb (e.g., on the BDD variable order). So if rule 2 uses the result of rule 1 and rule 1 does not use the result of rule 2, then put rule 1 and rule 2 in separate Datalog analyses.

5. Observe the sizes of the BDDs representing the relations that are input and output. bddbddb prints both the number of tuples in each relation and the number of nodes in the BDD. Try changing the BDD variable order for the domains of the relation, and observe how the number of nodes in the BDD for that relation change. You will notice that some orders perform remarkably better than others. Then note down these orders as invariants that you will not violate as you tweak other things.

6. The relative order of values *within* domains (e.g., in domains named `M`, `H`, `C`, etc. in Chord) affects the performance of bddbddb, but I've never tried changing this and studying its effect. It might be worth trying. For instance, John Whaley's PLDI'04 paper describes a specific way in which he numbers contexts (in domain `C`) and that it was fundamental to the speedup of his "infinity"-CFA points-to analysis.

7. Finally, it is worth emphasizing that BDDs are not magic. If your algorithm itself is fundamentally hard to scale, then BDDs are unlikely to help you a whole lot. Secondly, many things are awkward to encode as integers (e.g., the abstract contexts in domain `C` in Chord) or as Datalog rules. For instance, I've noticed that summary-based context-sensitive program analyses are hard to express in Datalog. The may-happen-in-parallel analysis provided in Chord shows a relatively simple kind of summary-based analysis that uses the Reps-Horwitz-Sagiv

tabulation algorithm. But this is as far as I could get—more complicated summary-based algorithms are best written in Java itself instead of Datalog.

15.3 BDD Representation

Each domain containing N elements is assigned log2(N) BDD variables in the underlying BDD factory with contiguous IDs. For instance, domain `F0` containing [128..256) elements will be assigned 8 variables with IDs (say) 63,64,65,66,67,68,69,70 and domain `Z0` containing [8..16) elements will be assigned 4 variables with IDs (say) 105,106,107,108.

If two domains are uninterleaved in the declared domain order in a Datalog program (i.e., '`_`' is used instead of '`x`' between them), then the BDD variables assigned to those domains are ordered in reverse order in the underlying BDD factory. For instance, the BDD variable order corresponding to the declared domain order `F0_Z0` is (in level2var format) "70,69,68,67,66,65,64,63,108,107,106,105".

If two domains are interleaved in the declared domain order in a Datalog program (i.e., '`x`' is used instead of '`_`' between them), then the BDD variables assigned to those domains are still ordered in reverse order of IDs in the underlying BDD factory, but they are also interleaved. For instance, the BDD variable order corresponding to the declared domain order `F0xZ0` is (in level2var format) "70,108,69,107,68,106,67,105,66,65,64,63".

Each BDD variable is at a unique "level" which is its 0-based position in the BDD variable order in the underlying BDD factory.

We will next illustrate the format of a BDD stored on disk (in a .bdd file) using the following example:

```
# V0:16 H0:12
# 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
# 82 83 84 85 86 87 88 89 90 91 92 93
28489 134
39 36 33 30 27 24 21 18 15 12 9 6 3 0 81 79 77 75 73 71 69 67 65 63 61 59 57 55  \
   53 51 82 80 78 76 74 72 70 68 66 64 62 60 58 56 54 52 37 34 31 28 25 22 19 16 \
   13 10 7 4 1 117 116 115 114 113 112 111 110 109 108 107 106 50 49 48 47 46 45 \
   44 43 42 41 40 105 104 103 102 101 100 99 98 97 96 95 94 93 92 91 90 89 88 87 \
   86 85 84 83 133 132 131 130 129 128 127 126 125 124 123 122 121 120 119 118   \
   38 35 32 29 26 23 20 17 14 11 8 5 2
287 83 0 1
349123 84 287 0
349138 85 0 349123
...
```

The first comment line indicates the domains in the relation (in the above case, `V0` and `H0`, represented using 16 and 12 unique BDD variables, respectively).

If there are N domains, there are N following comment lines, each specifying the BDD variables assigned to the corresponding domain.

The following line has two numbers: the number of nodes in the represented BDD (28489 in this case) and the number of variables in the BDD factory from which the BDD was dumped to disk. Note that the number of variables (134 in this case) is not necessarily the number of variables in the represented BDD (16+12=28 in this case) though it is guaranteed to be greater than or equal to it.

The following line specifies the BDD variable order in var2level format. In this case, the specified order subsumes `V0_H0` (notice that levels "81 79 77 75 73 71 69 67 65 63 61 59 57 55 53 51", which are at positions "14 15 ... 28 29" in the specified order are lower than levels "105 104 103 102 101 100 99 98 97 96 95 94" which are at positions "82 83 .. 92 93").

Each of the following lines specifies a unique node in the represented BDD; it has format "X V L H" where:

- X is the ID of the BDD node
- V is the ID of the bdd variable labeling that node (unless it is 0 or 1, in which case it represents a leaf node)
- L is the ID of the BDD node's low child
- H is the ID of the BDD node's high child

The order of these lines specifying BDD nodes is such that the lines specifying the BDD nodes in the L and H entries of each BDD node are guaranteed to occur before the line specifying that BDD node (for instance, the L entry 287 on the second line and the R entry 349123 on the third line are IDs of BDD nodes specified on the first and second lines, respectively).

Note on Terminology: The *support* of a BDD `b` is another BDD `r = b.support()` that represents all the variables used in `b`. The support BDD `r` is a linear tree each of whose nodes contains a separate variable, the low branch is 0, and high branch is the node representing the next variable. To list all the variables used in a BDD `b` use `r.scanSet()`. Needless to say, `scanSet()` simply walks along the high branches starting at the root of BDD `r`.