Chord: A Static and Dynamic Program Analysis Framework for Java

Mayur Naik

April 15, 2010

1 What is Chord?

Chord is a static and dynamic program analysis framework for Java. It has the following key characteristics:

- Stand-alone: various off-the-shelf program analyses are provided (e.g., may-alias, thread-escape, datarace, deadlock, etc.).
- Extensible: users can build their own program analyses on top of the provided ones.
- Compositional: each program analysis can be written independently and yet made to interact in complex ways with other program analyses.
- Efficient: results computed by each program analysis are cached for reuse by other program analyses without re-computation.
- Flexible: a broad range of program analyses can be expressed, including those written imperatively in Java or declaratively in Datalog, summary-based as well as cloning-based context-sensitive analyses, iterative refinement-based analyses, client-driven analyses, and combined static and dynamic analyses.

Chord is intended for use by a broad range of users:

- Program analysis writers: Program analysis researchers wanting to implement and evaluate new program analysis algorithms.
- Program analysis appliers: Researchers with possibly a limited program analysis background wanting to build applications on top of program analyses in Chord used as black boxes.
- Software engineers: Programmers wanting to use program analyses in Chord to assist with software development and testing.

Chord is intended to work on a variety of platforms, including Linux, Windows/Cygwin, and MacOS. It has been tested most extensively on Linux. It is open source software distributed under the New BSD License. Contributions from users are welcome and encouraged. The project website is located at http://code.google.com/p/jchord/.

2 Download and Installation

Ensure that the following software is installed on your machine:

- JDK 5 or higher, e.g. from IBM or Sun
- Apache Ant, a Java-based build tool
- a C++ compiler, e.g. GCC
- a Make utility, e.g. GNU Make
- Cygwin, if it is a Windows machine

Download a Chord source release from http://code.google.com/p/jchord/downloads/list or the latest source from the SVN repository at http://code.google.com/p/jchord/source/checkout.

Directory `main/` contains a `build.xml` file which is interpreted by Apache Ant. To see the various targets available, run the following command in that directory:

```
prompt> ant
```

To compile Chord, run the following command in the same directory:

```
prompt> ant compile
```

This will compile the following:

- the Java source code of Chord from `main/src/java/` to Java bytecode in `main/classes/`
- the C source code of BDD library BuDDy from `main/src/bdd/` to a shared library in `main/lib/` (`libbuddy.so` on Linux, `buddy.dll` on Windows, and `libbuddy.dylib` on MacOS); this library is needed for executing program analyses written in Datalog using bddbddb (a BDD-based Datalog solver).
- the C++ source code of the Chord instrumentation agent from `main/src/agent/` to a shared library in `main/lib/` (`libchord_instr_agent.so` on all architectures); this agent is needed for building program scope dynamically and for executing dynamic program analyses.

3 Setting up a Java Program

Suppose the program to be analyzed 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/`, the source and jar files of the libraries used by the program are under `lib/src/` and `lib/jar/`, respectively.

File `chord.properties` specifies properties to be passed to Chord (alternatively, these properties may be passed on the command-line, in format `-D<name>=<value>`). A sample such 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 path element in the value of any property named `chord.<...>.path` defined in this file is converted to an absolute path element with respect to the directory containing this file, which in the above case is `example/`.

Section 15 presents an exhaustive listing of properties recognized by Chord. Here, we only describe those defined in the above sample `chord.properties` file:

- `chord.main.class` specifies the fully-qualified name of the main class of the program.
- `chord.class.path` specifies the application-specific classpath of the program (the standard library classpath is implicitly included).
- `chord.src.path` specifies the Java source path of the program. All program 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 program 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 program 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` in the above example.

To run Chord on the above program, run the following command in Chord's `main/` directory:

```
prompt> ant -Dchord.work.dir=<...>/example run
```

where `<...>` denotes either the absolute path, or the path relative to Chord's `main/` directory, of the parent of directory `example/`. System property `chord.work.dir` specifies the working directory during Chord's execution.

The above command causes Chord to load the `chord.properties` file if present. The location of this file may be specified explicitly via property `chord.props.file` on the above command line. By default, it is `[chord.work.dir]/chord.properties`.

In the above case, Chord does not do much beyond loading the sample `chord.properties` file. For Chord to do something interesting, you need to set additional properties, either on the above command line or in the `chord.properties` file, specifying the task(s) Chord must perform. All tasks are summarized in Section 15.2. The two most common tasks, described next, are building the analysis scope of the given program (Section 4) and running program analyses on the given program (Section 5).

4 Building Analysis Scope

Chord computes the analysis scope (i.e., reachable classes and methods) 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 by calling method `chord.program.Program.v()`.

The algorithm used to compute the analysis scope is as follows.

- If property `chord.reuse.scope` has value `true` and the files specified by properties `chord.classes.file` and `chord.methods.file` exist, then Chord regards those files as specifying the classes and methods, respectively, to be regarded as reachable. The format of the classes file is a fully-qualified class name per line (e.g., `foo.bar.Main`). The format of the methods file is an entry of the form `<methodname>:<methoddescriptor>@<classname>` per line, specifying the method's name, the method's descriptor, and the method's declaring class (e.g., `main:([Ljava/lang/String;)V@foo.bar.Main`).

  The default value of property `chord.reuse.scope` is `false`, and the default values of properties `chord.classes.file` and `chord.methods.file` are `[chord.out.dir]/classes.txt` and `[chord.out.dir]/methods.txt`, respectively, where property `chord.out.dir` defaults to `[chord.work.dir]/chord_output/`, and property `chord.work.dir` defaults to the current working directory.

- If property `chord.reuse.scope` has value `false` or the file specified by property `chord.classes.file` or `chord.methods.file` does not exist, then Chord computes the analysis scope using the algorithm specified by property `chord.scope.kind` and then writes the classes and methods deemed reachable by that algorithm to those files.

  The possible legal values of property `chord.scope.kind` are [`dynamic`|`rta`|`rta_reflect`|`cha`]. In each case, Chord at least expects properties `chord.main.class` and `chord.class.path` to be set.

  - The `dynamic` value instructs Chord to compute the analysis scope dynamically, by running the program and observing using JVMTI 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 run-time overhead of JVMTI and to increase the predictive power of program analyses performed using the computed analysis scope.
  - The `rta` value instructs Chord to compute the analysis scope statically using Rapid Type Analysis (RTA). In this case, no attempt is made to resolve reflection.

    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.

RTA is a relatively inexpensive and precise algorithm in practice. Its key shortcoming is that it makes no attempt to resolve reflection, which is rampant in real-world Java programs, and can therefore be unsound (i.e., underestimate the set of reachable classes and methods). The next option attempts to overcome this problem.

- The `rta_reflect` value instructs Chord to compute the analysis scope statically using Rapid Type Analysis and, moreover, to resolve a common reflection pattern:

```
String s = ...;
Class c = Class.forName(s);
Object o = c.newInstance();
T t = (T) o;
```

  This analysis is identical to RTA except that it additionally inspects every cast statement in the program, such as the last statement in the above snippet, and queries the class hierarchy to find all concrete classes that subclass `T` (if `T` is a class) or that implement `T` (if `T` is an interface). Chord allows users to control which classes are included in the class hierarchy (see below).

- The `cha` value instructs Chord to compute the 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 (see below) and thereby control the precision and cost of CHA.

The default value of property `chord.scope.kind` is `rta`.

The class hierarchy is built if property `chord.scope.kind` has value `rta_reflect` or `cha`. Users can control which classes are included in building the class hierarchy by setting property `chord.ch.kind`, whose possible legal values are [`static`|`dynamic`]. 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. Chord's classpath, specified by property `chord.main.class.path`.
4. The classpath of user-defined Java analyses, specified by user-defined property `chord.java.analysis.path` (it is empty by default).
5. The classpath of the given program, specified by user-defined property `chord.class.path` (it is empty by default).

All classes in the entire classpath (resulting from items 1–5 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.
- All classes in Chord's classpath are excluded, i.e., all classes in the classpath specified by property `chord.main.class.path`, such as those in packages and sub-packages of `chord`, `joeq`, `net.sf.bddbddb`, `net.sf.javabdd`, `javassist`, `gnu.trove`, `net.sf.saxon`, etc.
- 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. By default, property `chord.ch.kind` has value `static`.
- 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 1–5 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.

5 Running a Program Analysis

TODO

6 Writing a Program Analysis

TODO

7 Writing a Dynamic Program Analysis

Follow the following steps to write your own dynamic analysis.

Step 1: Setting up the Instrumentation Scheme

Determine the instrumentation scheme required by your dynamic analysis, that is, the kind and format of events to be generated during an instrumented program's execution. See Section 8 for the kinds of supported events and their formats.

Step 2: Implementing the Analysis

Create a class extending `chord.project.DynamicAnalysis` and override the appropriate methods in it. The only method that must be compulsorily overridden is method `getInstrScheme()`, which must return an instance of the instrumentation scheme chosen in Step 1 above, plus each `process<event>(<args>)` method that corresponds to event `<event>` with format `<args>` enabled by the chosen instrumentation scheme. All other methods are no-ops if not overridden.

A sample such class called `MyDynamicAnalysis` is shown below:

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

// ***TODO***: analysis won't be recognized by Chord without this annotation
@Chord(name=<name-of-analysis>)

public class MyDynamicAnalysis extends DynamicAnalysis {
    InstrScheme scheme;

    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;
    }

    public void initAllPasses() {
        // ***TODO***: User code to be executed once and for all
        // before all instrumented program runs start.
    }

    public void doneAllPasses() {
        // ***TODO***: User code to be executed once and for all
        // after all instrumented program runs finish.
    }
```

```
    public void initPass() {
        // ***TODO***: User code to be executed once before each
        // instrumented program run starts.
    }

    public void donePass() {
        // ***TODO***: User code to be executed once after each
        // instrumented program run finishes.
    }

    // User-defined event handlers for this dynamic analysis.
    // No-ops if not overridden.
    public void process<event1>(<args1>) {
        // ***TODO***: User code for handling events of kind
        // <event1> with format <args1>.
    }
    ...
    public void process<eventN>(<argsN>) {
        // ***TODO***: User code for handling events of kind
        // <eventN> with format <argsN>.
    }
}
```

Step 3: Choosing or Implementing the Runtime Event Handler

Determine the runtime event handler for your dynamic analysis. There are two kinds of runtime event handlers, offline and online; see package `chord.runtime` for more details.

For most dynamic analyses, an offline event hander suffices, in which case you don't need to do anything in this step. Otherwise, you need to create a class extending `chord.runtime.Runtime` and override the appropriate methods in it, and provide the fully-qualified name of the created class as the value of property `chord.runtime.class` (whose default value is `chord.runtime.BufferedRuntime`).

Step 4: Compiling the Analysis

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

Step 5: Specifying Program Inputs

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.

Step 6: Running the Analysis

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

8 Instrumentation Events

Chord allows a given Java program to be instrumented to generate the following kinds of events during its execution.

1. Method entry and exit.
   Event: `ENTER_METHOD` m t
   Desc.: Generated after thread t enters method m respectively.
   Event: `LEAVE_METHOD` m t
   Desc.: Generated before thread t leaves method m respectively.

2. Loop entry and exit.
   Event: `ENTER_LOOP` w t
   Desc.: Generated before thread t enters loop w respectively.
   Event: `LEAVE_LOOP` w t
   Desc.: Generated after thread t leaves loop w respectively.

3. Object allocation
   Event: `BEF_NEW` h t
   Desc.: Generated only in the crude trace before thread t executes a `new` bytecode instruction at program point h.
   Event: `AFT_NEW` h t o
   Desc.: Generated only in the crude trace after thread t executes a `new` bytecode instruction at program point h and allocates new object o.
   Event: `NEW` h t o
   Desc.: Generated after thread t executes a `new` bytecode instruction at program point h and allocates new object o.
   Event: `NEW_ARRAY` h t o
   Desc.: Generated after thread t executes a `newarray` bytecode instruction at program point h and allocates new object o.

4. Getstatic primitive
   Event: `GETSTATIC_PRIMITIVE` e t f
   Desc.: Generated after thread t reads primitive-typed static field f at program point e.

5. Getstatic reference
   Event: `GETSTATIC_REFERENCE` e t f o
   Desc.: Generated after thread t reads object o from reference-typed static field f at program point e.

6. Putstatic primitive
   Event: `PUTSTATIC_PRIMITIVE` e t f
   Desc.: Generated after thread t writes primitive-typed static field f at program point e.

7. Putstatic reference
   Event: `PUTSTATIC_REFERENCE` e t f o
   Desc.: Generated after thread t writes object o to reference-typed static field f at program point e.

8. Getfield primitive
   Event: `GETFIELD_PRIMITIVE` e t b f

Desc.: Generated after thread t reads primitive-typed instance field f of object b at program point e.

9. Getfield reference
   Event: `GETFIELD_REFERENCE` e t b f o
   Desc.: Generated after thread t reads object o from reference-typed instance field f of object b at program point e.

10. Putfield primitive
    Event: `PUTFIELD_PRIMITIVE` e t b f
    Desc.: Generated after thread t writes primitive-typed instance field f of object b at program point e.

11. Putfield reference
    Event: `PUTFIELD_REFERENCE` e t b f o
    Desc.: Generated after thread t writes object o to reference-typed instance field f of object b at program point e.

12. Aload primitive
    Event: `ALOAD_PRIMITIVE` e t b i
    Desc.: Generated after thread t reads the primitive-typed element at index i of array object b at program point e.

13. Aload reference
    Event: `ALOAD_REFERENCE` e t b i o
    Desc.: Generated after thread t reads object o from the reference-typed element at index i of array object b at program point e.

14. Astore primitive
    Event: `ASTORE_PRIMITIVE` e t b i
    Desc.: Generated after thread t writes the primitive-typed element at index i of array object b at program point e.

15. Astore reference
    Event: `ASTORE_REFERENCE` e t b i o
    Desc.: Generated after thread t writes object o to the reference-typed element at index i of array object b at program point e.

16. Method call
    Event: `METHOD_CALL` i t
    Desc.: Generated before or after thread t executes the method invocation statement at program point i.

17. Thread start call
    Event: `THREAD_START` i t o
    Desc.: Generated before thread t calls the `start()` method of `java.lang.Thread` at program point i and spawns a thread o.

18. Thread join call
    Event: `THREAD_JOIN` i t o
    Desc.: Generated before thread t calls the `join()` method of `java.lang.Thread` at program point i to join with thread o.

19. Lock acquire
    Event: `ACQUIRE_LOCK` l t o
    Desc.: Generated after thread t executes a statement of the form monitorenter o or enters a method synchronized on o at program point l.

20. Lock release
    Event: `RELEASE_LOCK` r t o
    Desc.: Generated before thread t executes a statement of the form monitorexit o or leaves a method synchronized on o at program point r.

21. Thread wait call
    Event: `WAIT` i t o
    Desc.: Generated before thread t calls the `wait()` method of `java.lang.Object` at program point i on object o.

22. Thread notify call
    Event: `NOTIFY` i t o
    Desc.: Generated before thread t calls the `notify()` or `notifyAll()` method of `java.lang.Object` at program point i on object o.

9 Writing a Program Analysis in Datalog

TODO

10 Tuning a Program Analysis in Datalog

There are many tricks you can try to make `bddbddb` run faster:

1. Set system properties `noisy=yes`, `tracesolve=yes`, and `fulltracesolve=yes` while running `bddbddb` and observe which rule gets "stuck" (i.e. takes several seconds to solve). `fulltracesolve` is seldom useful, but `noisy` and `tracesolve` is 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 slowdown problem with that rule by either simplifying the relations involved (e.g., breaking their set of attributes into two if possible) or changing the relative ordering of the domains of those relations in `bddvarorder` (doing the latter alone frequently causes the problem to go away).

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 program, `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-program" of the Datalog program that seems problematic, and then focus on speeding up just that sub- program.

3. Tweak the BDD variable ordering in the Datalog file (the line that starts with `bddvarorder`). This is one of the most effective ways of making `bddbddb` run faster. As explained above, use `noisy` and `tracesolve` to find out which pairs of domains in the bddvarorder are responsible for the slowdown, and try to change their relative ordering (note that you can use either '_' or 'x' between a pair of domains, and the latter is commutative).

4. You can add the `.split` keyword at the end of certain rules as a hint to `bddbddb` to break up 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.

5. You can yourself break down rules by creating intermediate relations (the more relations you have on the RHS of a rule the slower it takes to solve that rule). This is another very effective way to make `bddbddb` run faster.

6. Try breaking down a single Datalog program into two programs. Of course, you cannot separate mutually-recursive rules into two different programs, but if you unnecessarily club together rules that could have gone into different programs, then they can put conflicting demands on `bddbddb` (e.g., on `bddvarorder`). 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 two different Datalog programs.

7. Observe the sizes of the BDDs representing the relations that are input and output. `bddbddb` tells both the number of tuples in each relation and the number of "nodes" in the BDD. Try changing the bddvarorder 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 orderings do remarkably better than others. Then note down these orderings as invariants that you will not violate as you tweak other things.

8. The relative ordering of values *within* domains (e.g. in domains named `M`, `H`, `C`, etc. in Chord) affects the solving time 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.

9. 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 the 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.

11 Program Representation

TODO

12 Points-to and Call-Graph Analyses

TODO

13 Datarace Analysis

Run the datarace analysis provided in Chord by running the following command in Chord's `main/` directory:

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

where directory `<...>` contains a file named `chord.properties` which defines properties `chord.main.class`, `chord.class.path`, and `chord.src.path`. See Section 15 for the meaning of these properties.

Directory `main/examples/datarace_test/` provides a toy Java program on which you 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 in Chord's `main/` directory with `<...>` replaced by `examples/datarace_test/`. Upon successful completion, the following files should be produced in directory `main/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.

14 Deadlock Analysis

Run the deadlock analysis provided in Chord by running the following command in Chord's `main/` directory:

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

where directory `<...>` contains a file named `chord.properties` which defines properties `chord.main.class`, `chord.class.path`, and `chord.src.path`. See Section 15 for the meaning of these properties.

Directory `main/examples/deadlock_test/` provides a toy Java program on which you 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 in Chord's `main/` directory with `<...>` replaced by `examples/deadlock_test/`. Upon successful completion, the file `deadlocks.html` should be produced in directory `main/examples/deadlock_test/chord_output/`.

15 Chord System Properties

Users can use system properties to control Chord's functionality. 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. Notation `[<...>]` is used in this section to denote the value of the property named `<...>`.

15.1 Basic Program Properties

This section describes basic properties of the given program that are required by program analyses in Chord, such as its main class, the location(s) of its class files and Java source files, and command-line arguments to be used to run the program.

- `chord.main.class`
  Type: class
  Description: Fully-qualified name of the main class of the given program (e.g., `com.example.Main`).
  Note: This property is required by virtually every task Chord performs.

- `chord.class.path`
  Type: path
  Description: Classpath of the given 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: `""`
  Note: This property is required by virtually every task Chord performs.

- `chord.src.path`
  Type: path
  Description: Java source path of the given 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 program analyses that need to present their analysis results at the Java source code level (by calling method `chord.program.Program.v().HTMLizeJavaSrcFiles()`).

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

- `chord.args.<id>`
  Type: string
  Description: Command-line arguments string to be used for the given program in the run having ID `<id>`.
  Default value: `""`
  Note: This property is used only when Chord executes the given program, namely, when it is

asked to build the analysis scope dynamically (i.e., when `[chord.scope.kind]=dynamic`) or when it is asked to execute a dynamic program analysis.

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

15.2 Chord Task Properties

This section describes properties that specify what task(s) Chord must perform, such as computing the analysis scope of the given program or running program analyses on the given program.

- `chord.build.scope`
  Type: bool
  Description: Compute the analysis scope (i.e., reachable classes and methods) of the given program using the algorithm specified by properties `chord.scope.kind` and `chord.reuse.scope`.
  Default value: `false`
  Note: The analysis scope is computed regardless of the value of this property if another task (e.g., a program analysis specified via property `chord.run.analyses`) demands it by calling method `chord.program.Program.v()`.

- `chord.run.analyses`
  Type: string list
  Description: List of names of program 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, and is the name of the class itself if this statement is missing. If the analysis is written in Datalog, its name is specified via a line of the form `"# name=<...>"`, and is the absolute location of the file itself if this line is missing.

- `chord.print.rels`
  Type: string list
  Description: List of names of program relations whose contents must be printed to files `[chord.out.dir]/<...>.txt` where `<...>` denotes the relation name.
  Default value: `""`
  Note: This task must be used with caution as certain program relations, albeit represented compactly as BDDs, may contain a large number (e.g., billions) of tuples, resulting in voluminous output when printed to a file. See Section 10 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.publish.targets`
  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 targets defined by program analyses in paths [chord.java.analysis.path] and [chord.dlog.analysis.path].
Default value: false

15.3 Chord Boot Properties

This section describes properties of the JVM running Chord, such as the working directory and various memory limits.

- chord.work.dir
  Type: location
  Description: Working directory during Chord's execution.
  Default value: current working directory

- chord.props.file
  Type: location
  Description: Properties file loaded by [chord.main.dir]/build.xml. Any of the below properties, as well as other user-defined properties to be passed to Chord (e.g., for user-defined analyses), may be set in this file. Each relative (instead of absolute) path element in the value of any property named chord.<...>.path set in this file is converted to an absolute path element with respect to the directory containing this file.
  Default value: [chord.work.dir]/chord.properties

- chord.max.heap
  Type: string
  Description: Maximum memory size of JVM running Chord.
  Default value: 1024m

- chord.max.stack
  Type: string
  Description: Maximum thread stack size of JVM running Chord.
  Default value: 32m

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

- chord.bddbddb.max.heap
  Type: string
  Description: Maximum memory size of JVM running bddbddb.
  Default value: 1024m
  Note: Each program analysis written in Datalog is run in a separate JVM because there may be multiple invocations of the Datalog solver bddbddb in a single run of Chord and it is difficult to reset the global state of bddbddb on each invocation.

15.4 Program Scope Properties

This section describes properties that specify how the analysis scope is computed. See Section 4 for more details.

- `chord.reuse.scope`
  Type: bool
  Description: Treat analysis scope as the classes and methods listed in files specified by properties `chord.classes.file` and `chord.methods.file`, respectively. Property `chord.scope.kind` is ignored if this property is set to `true` and the two files exist.
  Default value: `false`

- `chord.scope.kind`
  Type: [`dynamic`|`rta`|`rta_reflect`|`cha`]
  Description: Algorithm to compute analysis scope (i.e., reachable classes and methods). Current choices include `dynamic` (dynamic analysis), `rta` (static Rapid Type Analysis ignoring reflection), `rta_reflect` (static Rapid Type Analysis modeling a common reflection pattern), and `cha` (static Class Hierarchy Analysis).
  Default value: `rta`

- `chord.ch.dynamic`
  Type: bool
  Description: Compute the set of all classes loaded by the JVM while running the given program, and exclude classes not in this set (and their subclasses) while building the class hierarchy. This property is relevant only if `chord.scope.kind` is `rta_reflect` or `cha`, since only these two scope computing algorithms query the class hierarchy.
  Default value: `false`

- `chord.scope.exclude.std`
  Type: string list
  Description: List of prefixes of names of classes inside the standard library whose method bodies must be treated as no-ops by static scope building algorithms.
  Default value: `[chord.main.class.path]`

- `chord.scope.exclude.ext`
  Type: string list
  Description: List of prefixes of names of classes outside the standard library whose method bodies must be treated as no-ops by static scope building algorithms.
  Default value: `""`

- `chord.scope.exclude`
  Type: string list
  Description: List of prefixes of names of classes whose method bodies must be treated as no-ops by static scope building algorithms.
  Default value: `"[chord.scope.exclude.std],[chord.scope.exclude.ext]"`

15.5 Program Analysis Properties

This section describes properties regarding program analyses executed by Chord.

- `chord.java.analysis.path`
  Type: path
  Description: Classpath containing program analyses written in Java (i.e., `@Chord`-annotated classes).
  Default value: `[chord.main.dir]/classes/`

- `chord.dlog.analysis.path`
  Type: path
  Description: Path of directories containing program analyses written in Datalog (i.e., `*.datalog` and `*.dlog` files).
  Default value: `[chord.main.dir]/src/dlog/`

- `chord.reuse.rels`
  Type: bool
  Description: Construct program relations from BDDs stored on disk (from a previous run of Chord) whenever possible instead of re-computing them.
  Default value: `false`

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

- `chord.check.exclude.std`
  Type: string list
  Description: List of prefixes of names of classes and packages inside the standard library to be excluded by program analyses. Interpretation of this property is analysis-specific.
  Default value: `"sun.,com.sun.,com.ibm.jvm.,com.ibm.oti.,com.ibm.misc.,org.apache.harmony.,joeq.,jwutil.,java.,javax."`

- `chord.check.exclude.ext`
  Type: string list
  Description: List of prefixes of names of classes and packages outside the standard library to be excluded by program analyses. Interpretation of this property is analysis-specific.
  Default value: `""`

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

15.6 Program Transformation Properties

This section describes properties regarding program transformations performed by Chord.

- `chord.ssa`
  Type: bool
  Description: Do SSA transformation for all methods deemed reachable by the algorithm used to compute analysis scope.
  Default value: `true`

15.7 Chord Debug Properties

This section describes properties that specify the amount of debug information that Chord must produce during execution.

- `chord.verbose`
  Type: bool
  Description: Produce more verbose output during Chord's execution.
  Default value: `false`

- `chord.bddbddb.noisy`
  Type: bool
  Description: Produce more verbose output during the Datalog solver `bddbddb`'s execution.
  Default value: `false`

- `chord.save.maps`
  Type: bool
  Description: Write to file `[chord.bddbddb.work.dir]/<...>.map` when saving program domain named `<...>`.
  Default value: `true`
  Note: This functionality is useful for debugging Datalog programs using the `debug` target provided in file `build.xml` in Chord's `main/` directory (see Section 10).

15.8 Chord Instrumentation Properties

This section describes properties regarding execution of instrumented programs for dynamic analyses.

- `chord.reuse.trace`
  Type: bool
  Description: Reuse the dynamic trace file specified by property `chord.final.trace.file` (computed by a previous run of Chord) if it exists.
  Default value: `false`
  Note: Property `chord.trace.pipe` must be set to `false` if this property is set to `true`.

- `chord.trace.pipe`
  Type: bool
  Description: Implement the dynamic trace file as a POSIX pipe instead of a regular file.
  Default value: `true`
  Note: Property `chord.reuse.trace` must be set to `false` if this property is set to `true`.

- `chord.trace.block.size`
  Type: int
  Description: Number of bytes to read/write in a single operation from/to the dynamic trace file.
  Default value: `4096`

- `chord.runtime.class`
  Type: class
  Description: Subclass of `chord.project.Runtime` used by dynamic program analyses to handle events generated during an instrumented program's execution.
  Default value: `chord.project.BufferedRuntime`

- `chord.max.constr`
  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 `NEW` *h t o* to be generated (see Section 8). 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 the `BEF_NEW` *h t* events by `NEW` *h t o* and discards the `AFT_NEW` *h t o* events. 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 `NEW` *h i* 0 (i.e., it treats the created object as having ID 0, which is the ID also used for `null`).

15.9 Chord Output Properties

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

- `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.out.file`
  Type: location
  Description: Absolute location of the file to which the standard output stream is redirected during Chord's execution.
  Default value: `[chord.out.dir]/log.txt`

- `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: `[chord.out.dir]/log.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.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.bddbddb.work.dir`
  Type: location

Description: Absolute location of the directory used by the 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 JDK classes used by the given program are read/written by dynamic program analyses.
  Default value: `[chord.out.dir]/boot_classes/`

- `chord.user.classes.dir`
  Type: location
  Description: Absolute location of the directory from/to which instrumented non-JDK classes of the given program are read/written by dynamic program 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 dynamic trace files.
  Default value: `[chord.out.dir]/scheme.ser`

- `chord.crude.trace.file`
  Type: location
  Description: Absolute location of the crude dynamic trace file.
  Default value: `[chord.out.dir]/crude_trace.txt`

- `chord.final.trace.file`
  Type: location
  Description: Absolute location of the final dynamic trace file.
  Default value: `[chord.out.dir]/final_trace.txt`

15.10 Chord Resource Properties

This section describes properties specifying classpaths and locations of runtime libraries used by Chord. Ordinarily, users must not alter the default values of these properties.

- `chord.main.dir`
  Type: location
  Description: Absolute location of the `main/` directory in Chord's installation.

- `chord.lib.dir`
  Type: location
  Description: Directory containing libraries needed by Chord.
  Default value: `[chord.main.dir]/lib/`

- `chord.main.class.path`
  Type: path
  Description: Classpath of Chord. It includes the path specified by property `chord.java.analysis.path` to allow running user-defined program analyses.
  Default value: See `[chord.main.dir]/build.xml`

- `chord.bddbddb.class.path`
  Type: path
  Description: Classpath of `bddbddb`.
  Default value: See `[chord.main.dir]/build.xml`

- `chord.instr.agent.file`
  Type: location
  Description: Shared library implementing Chord instrumentation agent.
  Default value: `[chord.main.dir]/lib/libchord_instr_agent.so`

- `chord.javadoc.url`
  Type: string
  Description: URL of the Javadoc location of program analyses. It is used when publishing targets (i.e., when property `chord.publish.targets` is set to `true`).
  Default value: `http://chord.stanford.edu/javadoc/`

16 Acknowledgments

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

- Joeq, a Java compiler framework
- Javassist, a Java bytecode manipulation tool
- 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.