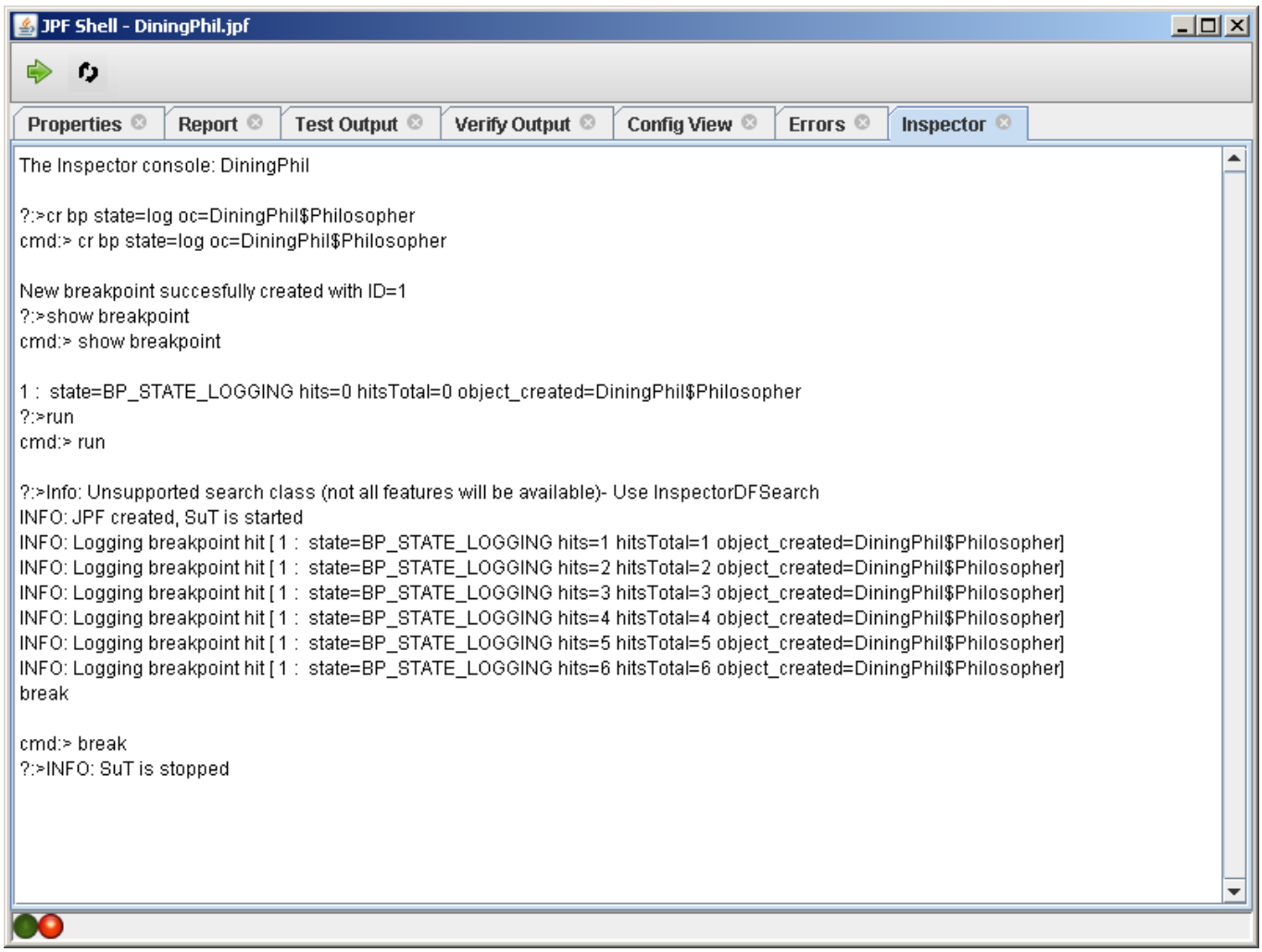
Inspector user guide

Main user interface of the JPF Inspector is a textual console that is embedded into a new shell panel. The console behaves in an expected way - a user writes textual commands and inspector prints results (command output). Current interface is shown on the figure below.



Enabling jpf-inspector

Inspector depends on jpf-shell so you need to have jpf-shell project installed as well. You have to define paths to `jpf-shell` and `jpf-inspector` projects in the configuration file (either *~/.jpf/site.properties* or **.jpf*). Then you should add `@using jpf-inspector` and `shell=` into the **.jpf* file you use.

Example: site.properties

```
jpf-base-dir  = /home/jancik/jpf/src
jpf-core      = ${jpf-base-dir}/jpf-core
jpf-shell     = ${jpf-base-dir}/jpf-shell
jpf-inspector = ${jpf-base-dir}/jpf-inspector
```

Example: *.jpf file

```
@using jpf-inspector
shell=.shell.basicshell.BasicShell

target = Racer
listener=gov.nasa.jpf.listener.PreciseRaceDetector
```

```
...
```

Breakpoints

Inspector supports five basic commands related to breakpoints. Each command has several alternative names - the main one is always followed by shortcuts in brackets. Each breakpoint has its own unique ID that is assigned automatically - the ID is printed when the breakpoint is created and by the `show bp` command.

The `run [continue]` command starts the execution of JPF on the SuT (instead of the Verify button) or continues with the execution (after the breakpoint was hit). The `break` command stops JPF immediately.

The `show breakpoint [show bp]` command prints all existing breakpoints and their status.

The `delete breakpoint [del bp]` command removes a breakpoint. Example: `del bp 10` (removes the breakpoint with ID 10).

The `create breakpoint [cr bp]` command is used to create new breakpoints.

- Each breakpoint may have a name that is not processed by the Inspector (`name = Identificator`).
- Each breakpoint must have a state that is defined in the creating command. Possible options are:
  - `disable [dis]` - the breakpoint is deactivated and only the hit counts are computed
  - `log` - if the breakpoint hits, then a log message appears in the console, but the SuT is not stopped
  - `enable [en]` - execution of JPF is stopped when the breakpoint hits

  Each breakpoint has 2 hit counters associated with it - a total hit counter and a path hit counter. A path hit counter behaves as hit counters in ordinary debuggers. Limits related to the associated action (log to console or stopping JPF) may be defined for each breakpoint. These limits are compared to the path hit counter. You may set either the lower or the upper bound, or both bounds. The keyword to use is `hit_count [hc]`. Some examples follow:
  - `10 <= hit_count` - an action is executed for the 10th hit, 11th hit, and so on
  - `hc < 50` - an action is executed for the initial 49 hits
  - `3 <= hc <= 3` - an action is executed only for the third hit of the breakpoint

Previous settings are common to all breakpoints and they have to take place before the breakpoint type specification in the `create breakpoint` command.

Breakpoint type specifications:

- `property_violated [pv]` - the breakpoint hits when some property is violated by SuT
- `position [pos] = file name : line number` - the breakpoint hits when the first instruction on the given line is executed. Example: `cr bp state=log pos=*/JPF/JPF.java:122`.
  Note: It is possible to use `*` in the `file name` argument as a wildcard for any part of the file path. This is especially useful to avoid writing directory (package) names - for example, you can write `*/ThreadStateError3.java:51` instead of `gov/nasa/jpf/rtembed/memory/ThreadStateError3.java`.
- `field_access [fa]`, `field_read [fr]` or `field_write [fw] = class name : field name` - the breakpoint hits if the given field of any instance of the given class is accessed (read, written). Example: `create breakpoint state=enable field_write=Racer:d`.
- `method_invoke [mi] = class name : method name` - the breakpoint hits if the given method of any instance of the given class is invoked. Example: `create breakpoint state=enable method_invoke=MyList:add`.
- `instruction_type [inst_type, it] = any / invoke [inv] / return [ret] / field_access [fa] / field_read [fr] / field_write [fw] / condition [cond] / lock / array` - the breakpoint hits if an instruction of the given type is executed. The `condition` type hits on the following instructions: IF_ACMP*, IF_ICMP*, IF*, IFNONNULL, and IFNULL. Bytecode instructions corresponding to other types are obvious (e.g., PUTFIELD matches `field_access` and `field_write`).
- `object_created [oc] = class name` - the breakpoint hits if the specified class (with the same name) is created. Example: `cr bp state=en 3<=hc<=3 object_created=java.lang.Thread` - breakpoint hits when the 3rd thread is created.
- `object_released [or] = class name` - the breakpoint hits if an object of the specified class is freed by the garbage collector. Example: `cr bp state=dis object_released=java.lang.String` - counts freed strings.
- `exception_thrown [et] = class name` - the breakpoint hits if an exception with the given type is thrown. Example: `cr bp state=en et=java.lang.RuntimeException` - hits when RuntimeException is thrown.
- `thread_scheduled [ts] = in / out / both : thread number` - the breakpoint hits when the specified thread is scheduled. The `: ThreadNumber` argument can be omitted, in this case the breakpoint hits if any thread not running in previous transition is scheduled. Example: `cr bp state=log ts=both:0` - hits if the main thread is scheduled in or out.

- `garbage_collection [gc] = begin / end / both` - the breakpoint hits if the garbage collection is started, stopped or in both cases. Example: `cr bp state=log garbage_collection=both`.
- `choice_generator [cg] = data / scheduling [sched] / both` - the breakpoint hits if a choice of the specified type is used (a new value is generated or a new thread is scheduled). Example: `cr bp stale=en 50<=hit_count choice_generator=data`.

A special breakpoint type is a *condition over program state*. The syntax is `variableID operator variableID`. Supported operators are ==, `!=`, <, <=, >, >=. Any expression that is a valid argument of the `print` command (see below) or a valid constant (true, false, null, “MyString”, '\n', 10.5, 1) can used as the `variableID`. Example: the condition `person.name == “John”` holds if the variable `person` has the field `name` and its value is `"John"`. The operator == works as in Java except for strings, for which it performs comparison of the actual strings (not references). It is also possible to use multiple conditions connected with propositional logic operators (`and`, `or`). Example: `cr bp i == 1 and val == null` is a valid breakpoint definition.

Dynamic assertions

It is possible to add simple assertions dynamically through the Inspector GUI. The command `assert position condition` creates a new assertion to the given source code `position` that checks the given `condition`. The `position` must have the form `file name : line number` and the `condition` can be any conditional expression over program state. Example: `assert Example02.java:35 j < 4` creates an assertion that will be violated (i.e., Inspector breaks its execution) if the condition `j < 4` does not hold at the first instruction associated with the line 35 in the file Example02.java.

Single stepping

Inspector supports single-step execution in a GDB-like manner. Currently it is possible to step over a source code line with or without skipping of nested method calls, and to step over an instruction - both in the forward and backward directions. All commands have a long name and a shortcut.

The default forward trace represents the longest program trace in the SuT which JPF Inspector recently traversed during backward stepping. It is a suffix of the current program trace that leads to the state where backward stepping has been started. The default forward trace should help users to find the state where JPF has been stopped and where the inspecting started.

The `step_over [so]` command tells JPF to execute all instructions on the current source code line, including all nested method calls.

The `step_in [si]` command tells JPF to execute all instructions on the current source code line, but to stop when a control steps in a nested method call.

The `step_instruction [sins]` command tells JPF to execute the next instruction and then stop.

The `step_out [sout]` command tells JPF to execute all instructions until the current method terminates.

The `step_transition [st] (scheduling [sched] | data | all*)` command tells JPF to execute all instructions until a choice generator of a given type is created (and the current transition terminates). The last executed instruction is from the terminated transition. You may use the command `enable ask choice_generators` (see below) to be able to specify the next choice before the next forward step.

The `back_step_over [bso]` command tells JPF to backtrack over all instructions on the current and previous source code lines, including all nested method calls.

The `back_step_in [bsi]` command tells JPF to undo all instructions on the current and previous source code lines. If the current method calls another method then the backtrack step stops before this call (instructions for pushing arguments of the nested method call on the stack are undone too). This command always undoes only a single method call.

The `back_step_instruction [bsins]` command tells JPF to undo the last instruction in the current trace.

The `back_step_out [bsout]` command tells JPF to undo all instructions that have been executed in the current method and stops at the begining of the source code line which calls the method.

The `back_step_transition [bst] (scheduling[sched] | data | all*)` command tells JPF to undo all instructions (from one or more transitions) until it backtracks over a choice generator of a given type. It stops just after the last instruction of the transition that precedes the CG.

Note that backward and forward steps (`so`/`bso` and `si`/`bsi`) are not fully complementary. The backward step over (command `back_step_over`) followed by the forward step over (`step_over`) reaches the original starting state only if the starting state is aligned to the first instruction on the current source code line. The same applies to other pairs of backward and forward stepping commands. If the starting state does not represent the first instruction on the given source code line (e.g., if the line is only partialy executed), then backward steps will backtrack over all executed instructions on the current line and undo also all instructions from the previous line. Subsequent forward step will terminate at the begining of next source line, i.e. at a different instruction than the instruction where the backward step started.

Be also aware of these two facts: (1) breakpoints are not disabled in the forward single-step execution and (2) breakpoint conditions are tested before each instruction is executed. Consequently, a step command may not move the program's execution forward if a breakpoint is hit on the current instruction (to be executed). On the other hand, breakpoints are ignored during backward steps.

Inspector also supports the mode in which the next choice at each state must be specified by the user (i.e., the user tells Inspector/JPF what choice from those provided by the current choice generator to use). This running mode is turned on by the command `enable (ask | print*) (scheduling | data | all*) choice_generators [cg]`, where terms in brackets are alternatives and the defaults are marked with `*`. It is disabled by the `disable` command whose arguments are the same as for `enable`. The first parameter tells Inspector whether to print the selected values (`print`) or ask the user for the choice (`ask`). The second parameter specifies what kinds of choice generators are considered. For example, the command `enable ask data cg` means that Inspector will ask for the next choice at each data choice generator, and the command `enable cg` only prints the choices (values) used. When Inspector asks for the next choice, the `cg select <index>` command can be used to specify its index. If the default forward trace exists then the choice (value) used in the trace is marked by `*`. Inspector prints the range of possible choices (indexes) and the description of each option. If the index is ommited then the default value is used. If a non-numeric value is selected then JPF stops execution just before the choice is selected and you can execute any other Inspector command. This can be used to disable asking for the choice. Important note: this works only if the `ChoiceGenerator.advance` method is **not** called outside of the core state space traversal procedure. In particular, calling this method in a custom listener breaks the user selection of next choices.

If a different value for the CG (different choice) than the value specified in the default forward trace is used, then the current default forward trace is discarded. If a breakpoint is reached and JPF execution is stopped, then the default forward trace is empty. If the user executes a backward step (over any choice generator), the default trace is created (extended).

The `used choice_generators [cg]` command prints all used choices. Result of this command is separated into two parts. The first part contains used choices in the current program trace. The optional second part represents the default forward trace, if it exists. The source code location associated with each choice is printed too.

Choices in the default forward trace are provided to the user, if the asking mode is enabled. Choices are used directly if the user does not specify values of the CG (e.g., when the CG print mode is used or during forward stepping).

Program state inspection

Inspector can print the current value of various components of a program state. Note that the Java program (upon which JPF is applied) must be compiled with debugging information for most of these commands to work. The # characters are needed to distinguish special keywords and the names of program variables.

The `thread [ti]` command prints the basic information (name, status, priority) of each thread. When a thread index is provided as an argument (optional), information is printed only for a single thread with the given index.

The `thread_pc [thpc]` command prints the source code line corresponding to the current program counter (current instruction) for all threads. When a thread index is provided as an optional argument, the source code line is printed only for a single thread with the given index.

The `print #thread[i]` command prints basic information about the i-th thread and its call stack (method names and source code lines). Information for the current thread is displayed, if no thread number is given.

The `print #thread[i].#stackFrame[j]` command prints all local variables (names, types, and values) in the j-th stack frame of the i-th thread.

The `print #thread[i].#stackFrame[j].var` command prints values of all instance fields of the object pointed to by the local variable `var` on the j-th stack frame of the i-th thread.

The `print #heap[i]` command prints values of all fields of a heap object with the index `i`. The index can be given in the decimal notation or in the hexa-decimal notation (0xNNNN). It is also possible to print all heap objects of a given type by using the `print #heap[class name]` command. The given class name can contain `*` to match any arbitrary number (including zero) of characters. This is useful to avoid writing package name. Example: `print #heap[*]` prints all objects on the heap and `print #heap[*.String]` prints all strings stored on the heap.

It is also possible to navigate through the heap via a chain of field names, and print values of all fields of corresponding objects. For example, commands like `print #thread[2].#stackFrame[1].#localVariable[1].left` and `print #heap[427].schedParams.allocTh` are valid. The filter `#static` restricts the view only on static fields of a given class. Instance fields are not reachable anymore. For example, the command `print #heap[100].#static` prints only values of static fields of the class, which is the type of the heap object with the index 100.

The command `print #heap[i].#super` prints values of all fields defined in the superclass of the dynamic class of the given heap object. The command `print #heap[i].#outerClass` prints values of fields defined in the outer (enclosing) class for the dynamic class of the heap object.

If the heap object with the index `i` is an array, then the command `print #heap[i][x]` can be used to access the element with the index `x`.

A few shortcuts are supported too. The `print` command with no arguments prints all local variables in the top stack frame of the current thread - it is a shortcut for `print #thread[i].#stackFrame[0]`, where `i` is the index of the current thread. The `print this` command prints values of all fields of the receiver object ("this") for the currently executed method.

Understanding output

We show the commands' output on the `DiningPhil` example from jpf-core.

Threads

Output of the `print #thread` command is

```
?:>print #thread[0]
cmd:> print #thread[0]
0 : main state=RUNNING priority=5
 DiningPhil.main(String[]) - DiningPhil.java:54 -     for (int i = 0; i < N; i++) {
```

where `0` is thread index and `main` is the thread's name. The call stack of the thread is printed. The `DiningPhil.main(String[])` fragment contains signature of a method on the call stack (class name, method name and parameter types). The `DiningPhil.java:54` fragment contains file name and source code line where the program counter is, and the `for (int i = 0; i < N; i++) {` fragment is the actual line from the source file.

Objects and methods

Output of the `print #thread[0].#stackFrame[0]` (or the `print` command without any parameters or `print #heap[315]`) looks like this:

```
?:>print
cmd:> print
DiningPhil.main(String[]) - DiningPhil.java:54 -     for (int i = 0; i < N; i++) {
       0 : args (java.lang.String[]) =[Ljava.lang.String;@137
       1 : forks (DiningPhil$Fork[]) =[LDiningPhil$Fork;@293
       2 : i (int) =5
```

Structure of the first line (`DiningPhil.main(String[]) - DiningPhil.java:54 - for (int i = 0; i < N; i++) {`) is the same as in the previous example. Then follows an indented list of local variables and parameters (in the case of a method) or fields (in the case of objects). Take the first variable (`0 : args (java.lang.String[]) =[Ljava.lang.String;@137`) - 0 is the index of the variable (which can be used in subsequent print commands), `args` is the variable name, `java.lang.String[]` is the type of a variable, and `[Ljava.lang.String;@137` is the variable's value. For variables with primitive types, the value is printed directly (see the 3rd entry with the `int` type). For variables with reference types (objects or arrays), the value consists of the actual type (in the internal format, e.g. `[Ljava.lang.String;`) followed by `@` and the index of an object on the heap to which the reference points. It is then possible to use the `print #heap[137]` command to see the internal structure of an object (to follow the reference).

Program state modification

An important debugging feature is program state modification. It allows the user to change the values of some fields and local variables, and then check how the changes influence program behavior - in particular, whether the given error still occurs or not.

The `set expr = value` command can be used to assign a new value to local variables, method parameters and objects fields. It is possible to set values even for *static* and *final* variables. The variables to be changed are identified by the same expressions as in the `print` command, i.e. they have the same syntax. The new value can be another variable (auto-unboxing is supported) or a constant of the proper type (see the examples below).

Important notes

- Conversion between variables and values of the types *byte*, *short*, *int*, *long* is automatic and the ranges are checked.
- Lossless conversion from *float* to *double* is automatic, but the other direction is not supported.
- To assign a new *float* constant, use the suffix 'f' after the actual constant (example: *10.1f*).
- To assign large numbers to a *long* variable, use the suffix 'L' after the constant (example: *10L*).

Examples

- `set my_bool = true`, `set myBool = false` - assigns constant to a boolean variable.
- `set my_charArray[10] = 'A'` - assigns the character 'A' to the char array at the index 10.
- `set my_byte = my_Integer` - assigns the current value of the existing variable `my_Integer` of the `java.lang.Integer` type to a byte variable.
- `set my_const_Object = null` - assigns `null` value to the given reference variable (of any type).
- `set my_String = "Alf rules"` - assigns the given string constant to a string variable.
- `set #heap[10].float_val = 3.1415f` - assigns the Pi value defined as a float constant to the field `float_val` of the specified heap object.
- `set my_double = NaN` - the user can assign special values `NaN` (`not-a-number`), `-inf` (`negative_infinity`), and `+inf` (`positive_infinity`) to floating point variables.

Record and replay

Inspector allows to save (record) all commands executed in the current session into a given file and replay (re-execute) them later in a batch manner.

The `record [rec] save filename` command saves the current session (all executed commands) into the file.

The `rec clear` command clears all recorded information.

The `rec print` command prints all recorded information to the console. Output contains "special commands" with the prefix `callback_`, which cannot be written directly into the prompt.

The `rec execute [ex] filename` command loads the given file and executes all commands recorded in the file. Any occurrence of the `rec save` command in the replay/execute mode is ignored.

Inspector API

It is also possible to call Inspector from other programs. Test infrastructure shows one possible way how it can be done. However, this "API" is not officially supported yet and can be changed at any moment.