Improving the BudgetChecker Listener Report 2

Connor Ahearn, Jeremy Winkler

April 20th, 2020

1. Goals

The listener BudgetChecker is used for keeping track of the resource budget for the program during execution. Its capable of letting you know when:

- Time is exceeded
- Heap is exceeded
- Depth is exceeded
- Number of states are exceeded
- Number of new states are exceeded

For example, if we set a property `budget.max_state=5`, then if the program creates more than five states, the execution stops and returns the message: `max states exceeded: 5`

2. Changes we made

Short summary of what we have done for the class.

1. Documentation of the class - Javadoc
   Previously, none of the methods had associated Javadoc and the possible parameters were never indicated anywhere in the Javadoc. After the refactor, each method has its own Javadoc and the class Javadoc now explains the different parameters that can be used. The changes can be found in 4.1 Code Refactor.

2. Describe how to use the listener
   A basic user guide for the listener can be found in the readme submitted. It lists the properties for the listener and goes over a simple example.

3. Refactoring code to improve readability where applicable
   Although most of the code was relatively well written, there were certain things which we still had to change. Some of the changes we made were:
   - Cryptic variable names
   - Convoluted conditions
   - Magic numbers

Once again, the changes we made can be found in 4.1 Code Refactor.

4. Develop JPF tests

For testing, our general idea was to make two tests for each possible budget constraint. One for the budget constraint not being exceeded and one where it was exceeded. After this, we decided that we had two more groups of tests to do. One group was for testing multiple properties at once. Our process for this was to take a few properties and for that set of properties, make one test where each of these properties fails once. We ensured that our sets covered each property failing at least once. Our last group of testing to ensure that the listener still works for concurrent programs. Concurrency should work out of the box considering it is dealt with by JPF, but we tested this just to make sure. For each test that was checking the budget constraints separately, we added a duplicate version of it which ran some concurrent method and updated the parameters as needed.

To implement the tests, we use the following skeleton presented during one of the lectures.

```
resetProperties();
// Set required properties ...
PrintStream out = System.out;
ByteArrayOutputStream stream = new ByteArrayOutputStream();
System.setOut(new PrintStream(stream));

if(this.verifyNoPropertyViolation(PROPERTIES)) {
	...
} else {
	System.setOut(out);
	...
}
```

where the constant is defined by

```
private static String[] PROPERTIES = { "+classpath=./bin",
                                       "+native_classpath=./bin",
                                       "+listener=BudgetChecker",
                                       "+cg.enumerate_random = true",
                                       "",
                                       "",
                                       "",
                                       "",
                                       "",
                                       "",
                                       "" };
```

In total, we developed 35 tests.

5. New property

Although this change was not as large as the others listed here, we decided to add this since it did not really fit in with any of the other topics. The property that we added was called `check_interval` which determines how frequently the listener checks the properties. This could be useful if more precise checks are needed for any reason.

3. Potential Future Improvements

If we were to continue to work on this project, the logical next step would be further testing in an effort to improve the documentation. We greatly improved the documentation from where we found it at the start, however some of the parameters passed in the config file are a bit vague, and after working with the listener even we weren't quite sure what the difference was between max_new_states and max_states, and when you would want to use one over another.

The idea behind further documentation would ideally lead towards an actual user manual with examples and explanations of the services made available by the class, helping users take advantage of the features it makes available.

4. Appendix

4.1 Code Refactor

Changes made are color coded using the following color scheme:

- Documentation
- Variable Names
- Magic Numbers
- Other

Things grouped under other include:

- New property
- Private variables
- Convoluted if statement
- Initialize variables

```
/*
 * Copyright (C) 2014, United States Government, as represented by the
 * Administrator of the National Aeronautics and Space Administration.
 * All rights reserved.
 *
 * The Java Pathfinder core (jpf-core) platform is licensed under the
 * Apache License, Version 2.0 (the "License"); you may not use this file except
 * in compliance with the License. You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0.
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
```

```
 * limitations under the License.
 */
//package gov.nasa.jpf.listener;

import gov.nasa.jpf.Config;
import gov.nasa.jpf.JPF;
import gov.nasa.jpf.ListenerAdapter;
import gov.nasa.jpf.annotation.JPFOption;
import gov.nasa.jpf.annotation.JPFOptions;
import gov.nasa.jpf.report.Publisher;
import gov.nasa.jpf.search.Search;
import gov.nasa.jpf.vm.Instruction;
import gov.nasa.jpf.vm.ThreadInfo;
import gov.nasa.jpf.vm.VM;

import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;
import java.lang.management.MemoryUsage;

/**
 * Listener that implements various budget constraints
 */
@JPFOptions({
  @JPFOption(type = "Long", key = "budget.max_time", defaultValue= "-1", comment =
"stop search after specified duration [msec]"),
  @JPFOption(type = "Long", key = "budget.max_heap", defaultValue = "-1", comment=
"stop search when VM heapsize reaches specified limit"),
  @JPFOption(type = "Int", key = "budget.max_depth", defaultValue = "-1", comment =
"stop search at specified search depth"),
  @JPFOption(type = "long", key = "budget.max_insn", defaultValue = "-1", comment =
"stop search after specified number of intstructions"),
  @JPFOption(type = "Int", key = "budget.max_state", defaultValue = "-1", comment =
"stop search when reaching specified number of new states"),
  @JPFOption(type = "Int", key = "budget.max_new_states", defaultValue = "-1",
comment= "stop search after specified number of non-replayed new states"),
  @JPFOption(type = "Int", key = "budget.check_interval", defaultValue = "-1",
comment= "decides how often the checks within instructionExecuted happen")
})

/**
 * The BudgetChecker listener is designed to treat the resources of the
 * local machine JPF is running on as a set of constraints with set values
 * (or a "budget") that shouldn't be exceeded during a JPF search.
 *
 * Constraints for the BudgetChecker have to be set within the .jpf configuration
 * file. Options for these are as follows:
 *
 * budget.max_time -- This sets the max amount of time in miliseconds that the
search should run
 * budget.max_heap -- This is the upper limit on how large the search heap can be
 * budget.max_depth -- This is the upper limit on how deep the search can go
 * budget.max_insn -- This is the upper limit on the number of instructions that
the search will run
 * budget.max_state -- This is the upper limit on new states reached in the search
```

```
 * budget.max_new_states -- This is the upper limit on new states that are not a
trace replay reached in the search
 * budget.check_interval -- This defines how often the checks within
instructionExecuted are run. By default it is 10,000
 *
 * Unless specified, the default value for the options listed are 0. Value of 0
indicates that the budget checker will not be checked.
 */
public class BudgetChecker extends ListenerAdapter {

  private long startTime;
  private MemoryUsage memoryUsage;
  private long startingMemoryUsage;
  private MemoryMXBean memoryBean;

  // Standard Listener fields set by constructor during initialization
  private VM vm;
  private Search search;

  // Global Counters
  private long insnCount;
  private int newStates;

  // the budget thresholds

  /**
   * The max time the model will be allowed to execute.
   * This field is set in the config file through budget.max_time
   * If not set, this field is ignored
   */
  private long maxTime;

  /**
   * The maximum size that the heap will be allowed to reach in the model.
   * This field is set in the config file through budget.max_heap
   * If not set, this field is ignored
   */
  private long maxHeap;

  /**
   * The maximum size that the depth of the search will be allowed to reach in the
model.
   * This field is set in the config file through budget.max_depth
   * If not set, this field is ignored
   */
  private int maxDepth;

  /**
   * The maximum amount of instructions ran the search will be allowed to reach in
the model.
   * This field is set in the config file through budget.max_insn
   * If not set, this field is ignored
   */
  private long maxInsn;
```

```
  /**
   * The maximum amount of states the search will be allowed to reach in the model.
   * This field is set in the config file through budget.max_state
   * If not set, this field is ignored
   */
  private int maxState;

  /**
   * The maximum amount of new states the search will be allowed to reach in the
model.
   * This field is set in the config file through budget.max_new_state
   * If not set, this field is ignored
   */
  private int maxNewStates;

  /**
   * The number of instructions to be executed before the instruction count is
checked
   * This field is set in the config file through budget.check_interval
   * If not set, this field is set to a default 10,000 instruction interval
   *
   * See instructionExecuted (VM vm, ThreadInfo ti, Instruction nextInsn,
Instruction executedInsn) for more
   */
  private int checkInterval;

  // the message explaining the exceeded budget
  private String message;

  /**
   * Initializes a new BudgetChecker Listener object for the
   * corresponding JPF instance and configuration file
   * passed to it.
   * @param conf Information contained in the configuration file
   * @param jpf Access to the JPF instance that is running
   */
  public BudgetChecker (Config conf, JPF jpf) {
    // initialize counters
    newStates = 0;
    insnCount = 0;

    //--- get the configured budget limits (0 means not set)
    maxTime = conf.getDuration("budget.max_time", 0);
    maxHeap = conf.getMemorySize("budget.max_heap", 0);
    maxDepth = conf.getInt("budget.max_depth", 0);
    maxInsn = conf.getLong("budget.max_insn", 0);
    maxState = conf.getInt("budget.max_state", 0);
    maxNewStates = conf.getInt("budget.max_new_states", 0);
    checkInterval = conf.getInt("budget.check_interval", 10000);

    startTime = System.currentTimeMillis();

    if (maxHeap > 0) {
```

```
    memoryBean = ManagementFactory.getMemoryMXBean();
    memoryUsage = memoryBean.getHeapMemoryUsage();
    startingMemoryUsage = memoryUsage.getUsed();
  }

  search = jpf.getSearch();
  vm = jpf.getVM();
}

/**
 * Method that checks if the time that the search has ran has
 * exceeded the max time specified in the configuration file.
 *
 * @return true if the time has exceeded, false otherwise
 *     - If budget.max_time is not set, returns false
 */
public boolean timeExceeded() {
  if (maxTime > 0) {
    long duration = System.currentTimeMillis() - startTime;
    if (duration > maxTime) {
      message = "max time exceeded: " + Publisher.formatHMS(duration)
             + " >= " + Publisher.formatHMS(maxTime);
      return true;
    }
  }

  return false;
}

/**
 * Method that checks if the amount of space taken by the heap
 * has exceeded the size specified in the configuration file.
 *
 * @return true if the heap has exceeded, false otherwise
 *     - If budget.max_heap is not set, returns false
 */
public boolean heapExceeded() {

  if (maxHeap > 0) {

    // Constant used for the amount of bytes in a megabyte
    final int MEGABYTE = 1048576;

    MemoryUsage memoryUsage = memoryBean.getHeapMemoryUsage();
    long used = memoryUsage.getUsed() - startingMemoryUsage;
    if (used > maxHeap) {
      message = "max heap exceeded: " + (used / MEGABYTE) + "MB"
                   + " >= " + (maxHeap / MEGABYTE) + "MB" ;
      return true;
    }
  }

  return false;
}
```

```
/**
 * Method that checks if the depth of the search has exceeded
 * the limit specified in the configuration file
 *
 * @return true if the depth has exceeded, false otherwise
 *     - If budget.max_depth is not set, returns false
 */
public boolean depthExceeded () {
  if (maxDepth > 0) {
    int depth = search.getDepth();
    if (depth > maxDepth) {
      message = "max search depth exceeded: " + maxDepth;
      return true;
    }
  }

  return false;
}

/**
 * Method that checks if the number of instructions ran
 * has exceeded the limit specified in the configuration file
 *
 * @return true if the instruction count has exceeded, false otherwise
 *     - If budget.max_insn is not set, returns false
 */
public boolean insnExceeded () {
  if (maxInsn > 0) {
    if (insnCount > maxInsn) {
      message = "max instruction count exceeded: " + maxInsn;
      return true;
    }
  }
  return false;
}

/**
 * Method that checks if the number of states reached has
 * exceeded the limit specified in the configuration file
 *
 * @return true if the state count has exceeded, false otherwise
 *     - If budget.max_state is not set, returns false
 */
public boolean statesExceeded () {
  if (maxState > 0) {
    int stateId = vm.getStateId();
    if (stateId >= maxState) {
      message = "max states exceeded: " + maxState;
      return true;
    }
  }

  return false;
```

```
}

/**
 * Method that checks if the number of new states found has
 * exceeded the limit specified in the configuration file
 *
 * @return true if the new states count has exceeded, false otherwise
 *     - If budget.max_new_states is not set, returns false
 */
public boolean newStatesExceeded(){
  if (maxNewStates > 0){
    if (newStates > maxNewStates) {
      message = "max new state count exceeded: " + maxNewStates;
      return true;
    }
  }
  return false;
}

/**
 * Anytime the state advances, this method checks if
 * the time, heap, state count, depth or new state count
 * have exceeded their limits. If they have, the search
 * terminates and a message describing why is passed on
 * to the JPF report
 *
 * @param search Search object corresponding to the current search thats running
 */
@Override
public void stateAdvanced (Search search) {
  if (timeExceeded() || heapExceeded()) {
    search.notifySearchConstraintHit(message);
    search.terminate();
  }

  if (search.isNewState()){
    if (!vm.isTraceReplay()){
      newStates++;
    }
    if (statesExceeded() || depthExceeded() || newStatesExceeded()){
      search.notifySearchConstraintHit(message);
      search.terminate();
    }
  }
}

/**
 * This method checks instruction based budget checks on instruction counts
 * that correspond to the budget.check_interval parameter in the
 * jpf config file. By default, every 10,000 instructions.
 *
 * This method checks if
 * the time, heap size or amount of instruction executed has
 * exceeded their limits. If they have, the search terminates
```

```
   * and a message why is passed on to the JPF report
   *
   * @param vm JPF VM related to the current model check
   * @param threadInfo Thread information provided by JPF required for providing
instruction statistics & thresholds
   * @param nextInsn Instruction that will run next
   * @param executedInsn Instruction that has just run
   */
  @Override
  public void instructionExecuted (VM vm, ThreadInfo threadInfo, Instruction
nextInsn, Instruction executedInsn) {

    // Checks every CHECK_INTERVAL instructions excecuted
    insnCount++;

    if ((insnCount % checkInterval) == 0) {
      if (timeExceeded() || heapExceeded() || insnExceeded()) {
        search.notifySearchConstraintHit(message);

        vm.getCurrentThread().breakTransition("budgetConstraint");
        search.terminate();
      }
    }
  }

}
```