LTL Verification in JPF

Chung-Tuyen Luu, Dinh-Phuc Nguyen, Ewgenij Starostin.
Mentors: Franco Raimondi, Hoang Truong

Introduction

This is the result of two projects:

- *“Construction of linear temporal property verification extension”*
- *“LTL verification in JPF”*

Aim: *Enable the verification of LTL properties in Java code*.

Issues:

- Define syntax of LTL formulae appropriately.
- Runs could be either finite or infinite.
- Previous work based on listeners, but a new search strategy (Double DFS) is needed for infinite traces.
- Investigate interaction with symbolic execution.

LTL syntax: operators

Operators:

- Boolean: `/\` (`&&`), `\/` (`||`), `->` and `!`
- Temporal: `[]`, `<>`, `U`
- Additionally: `W` (weak until), `V` (release), `M` (weak release)

- Formulae are read by a custom parser.
- The parser uses a modified version of **ltl2buchi** (by Flavio and Dimitra) to construct a Buchi automaton.

LTL syntax: atoms

Atoms can be:

- Boolean variables, e.g.
  - `packageName.ClassName.field` and
  - `packageName.ClassName.method(T).var`
- An expression, such as `(x+y)*z - 3.0 > u/5` (where x,y,z,u are written as above).
  - Evaluated by doing the arithmetics, or false if a variable isn't available in the current state.
- A method name, e. g. `package.Class.methodName(int,float,T)`
  - True if the current instruction is an `InvokeInstruction` for `package.Class.methodName(int,float,T)`.

LTL syntax: examples

Using annotation-like style:

```
import gov.nasa.jpf.ltl.LTLSpec;
@LTLSpec("[](Simple.f2(int,int)->
             X(<>Simple.f1(String)))")
public class Simple { [...] }
```

Other examples:

- `@LTLSpec("[](<>Test.done() && <>Test.foo())")`
- `@LTLSpec("Test2.i==0 U Test2.test(int).a>8")`
- From a file: `@LTLSpecFile("spec.ltl")`

Finite traces

- Create a finite automaton for the LTL formula (see “Automata-Based Verification of Temporal Properties on Running Programs” by Dimitra and Klaus).
- Transitions in the automaton are labelled with atoms.
- An auto-loading listener performs the verification.
- After executing every instruction:
    - Extract the values of variables in the LTL formula.
    - Check enabled transitions from current state in automaton.
    - Report violation if automaton cannot make progress.
    - Report violation if one path terminates in a non-accepting state.

Finite traces: symbolic checking

An atom in a transition guard is symbolic if it references a symbolic value.

Checking procedure:

- Convert all variables of the atom to symbolic values.
- The atom is satisfiable iff it translates to a constraint which is satisfied in the current symbolic state of the program.
- The symbolic state of the SUT is represented by the path condition, which is also a symbolic constraint.

Finite traces: symbolic execution

We add the atom constraint to the path condition and check if the new constraint is satisfiable.
→ The path condition changes when checking symbolic atoms in a transition guard.

In detail:

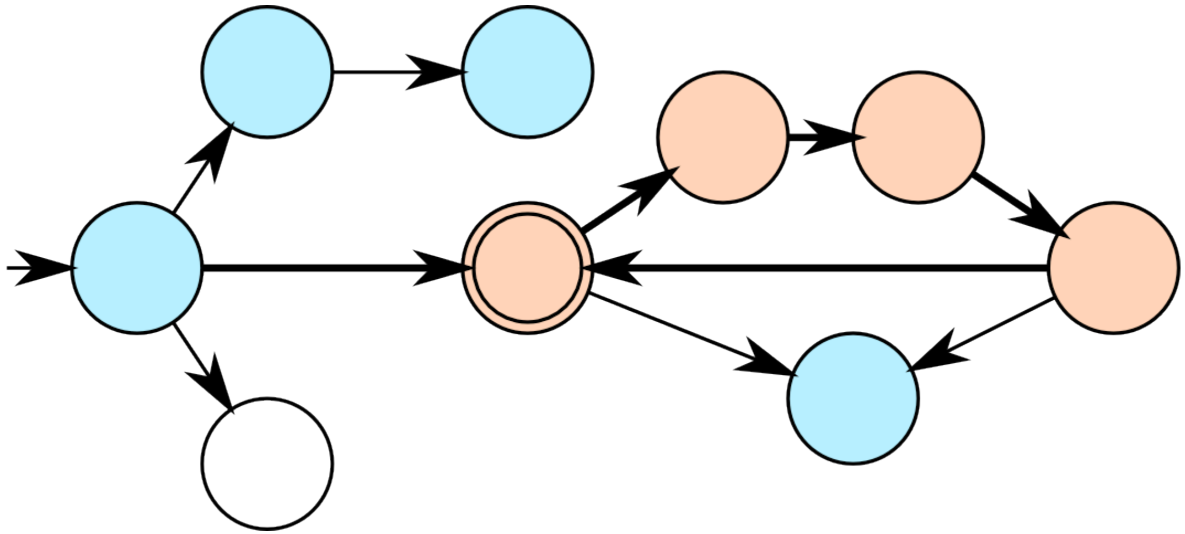
- Create an `AutomataChoiceGenerator` (extending `PCChoiceGenerator`), overriding `getCurrentPC()` method to return the corresponding path condition for each branch.
- If appending the guard to the PC adds any branches, this CG is registered as the next `ChoiceGenerator`.

Infinite executions

Principle:

- Verification is performed using double depth first search on the product of system and generated Buchi automaton for the negated specification.
- DDFS looks for a cycle in the product graph that contains an accepting Buchi state, i. e. a way for the system to violate the specification.
    - DFS phase 1: find path to accepting Buchi state.
    - DFS phase 2: find a cycle from that state back to itself.
- DDFS is implemented as a new search strategy in JPF.

Infinite executions: details



- `gov.nasa.jpf.ltl.ddfs.DDFSearch` keeps track of two sets of visited states: DFS2 ⊆ DFS1.
- Currently using `FullStateSet` for state matching.
- Buchi automaton states are tracked in a queue in `DDFSearch` (not visible for JPF).
- A listener is used to step through the system in single VM instructions.

Installation

Standard installation procedure:

```
$ hg clone https://bitbucket.org/francoraimondi/
jpf-ltl
$ ant build
```

- Approximately 20 examples available under src/examples

How to use it

- Add LTL property in .java file (or in an external file).
- .jpf for infinite traces (set `finite=false` and the search and storage classes):

```
target = YourClass
finite=false
search.class=gov.nasa.jpf.ltl.ddfs.DDFSearch
vm.storage.class=gov.nasa.jpf.jvm.FullStateSet
```

- .jpf for finite traces (set `finite=true`):

```
@using jpf-symbc
target = YourClass
finite=true
```

Example: Dining Philosophers

```
import gov.nasa.jpf.ltl.LTLSpec;
@LTLSpec ("[]!(
    Dining.waiting[0] && Dining.waiting[1] &&
    Dining.waiting[2] && Dining.waiting[3] &&
    Dining.waiting[4])")
public class Dining {
  static boolean[] waiting = new boolean[5];
  [...]
  public static void main (String[] args) {
    for (int i = 0; i < 5; i++)
      new Thread () { [...] }.start ();
    // This will never deadlock:
    while (true) Thread.yield ();
  }
}
```

Example where the Buchi automaton is not equal to the FSA:

```
import gov.nasa.jpf.ltl.LTLSpec;
@LTLSpec("[](<>Test.done() && <>Test.foo())")
public class Test {
    public static void main(String[] args) {
        int y = 11;   // y = 7 would be finite
        int x = 0;
        while (x != y) {
            x = x+1;
            if (x > 9) x = 0;
            done();
        }
        foo();
    }
    public static void done() {}
    public static void foo() {}
}
```

Future work

- Refactor code to minimize duplicated code between finite and infinite branches.
- Add symbolic execution to DDFS (currently being tested).
- Add mixed finite/infinite Buchi automata and modify DDFS (theoretical work in progress).
- Provide set of tests (currently being developed).
- Improve documentation (we are working on it!).