Using Java Pathfinder to Reason about Agent Systems

Franco Raimondi

f.raimondi@mdx.ac.uk
Department of Computer Science
Middlesex University
http://www.rmnd.net

Liverpool, 11th September 2015

Joint work with...

Joint work with a number of people. In particular:

- Neha Rungta at NASA Ames.
- G. Brat, C. Cardoza, W. Clancey, M. Goodrich, J. Holbrook, J. Hunter, E. Mercer, G. Primiero, M. Shafto, R. Stocker.

Software, news, (some) tutorials and publications available at:

- `http://www.rmnd.net`
- `http://mase.cs.mdx.ac.uk`

"Real" applications

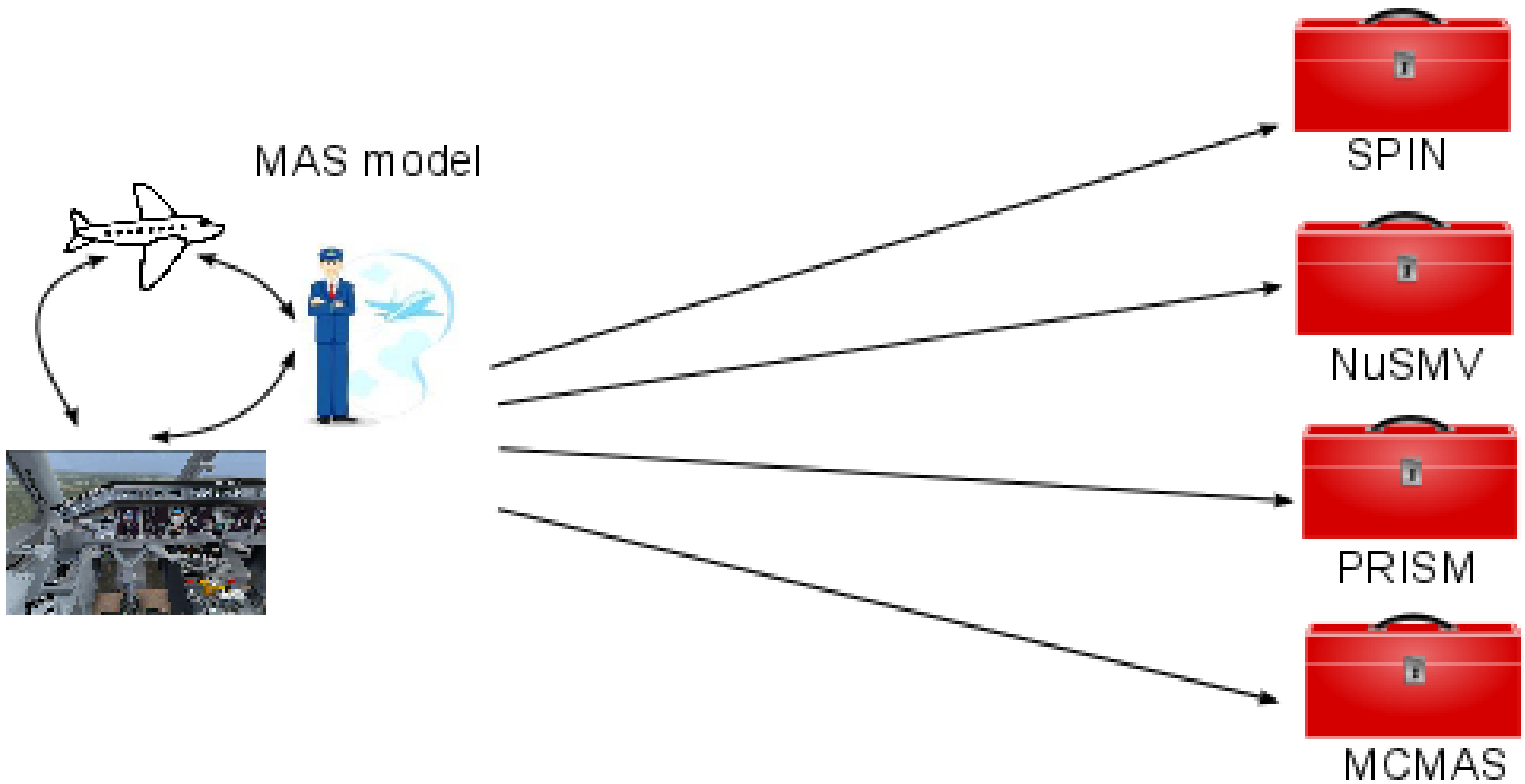
- Various scenarios are available
- Developers and engineers would like to use MAS verification (for autonomous systems etc.)

BUT *"I cannot translate my code to ISPL!"* is a very common remark.

It's not a problem with ISPL only. My other attempts:

- A. Lomuscio, C. Pecheur, F. Raimondi, *Verification of knowledge and time with NuSMV* (based on C. Pecheur and F. Raimondi, *Symbolic model checking of logics with Actions*)
- F. Raimondi, C. Pecheur, A. Lomuscio, *Applications of model checking for multi-agent systems: verification of diagnosability and recoverability.*

Current situation



This picture can be modified by using JPF...

Short Tutorial: Java Pathfinder

- JPF is a popular "model checker" for Java code. In its default configuration JPF detects unhandled exceptions, deadlocks, and races.
- JPF is essentially a customizable JVM.

`http://jpf.byu.edu/`

The notion of JPF *state* is important! I need some preliminaries...

Java bytecode generation + execution overview

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

Java bytecode generation + execution overview

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

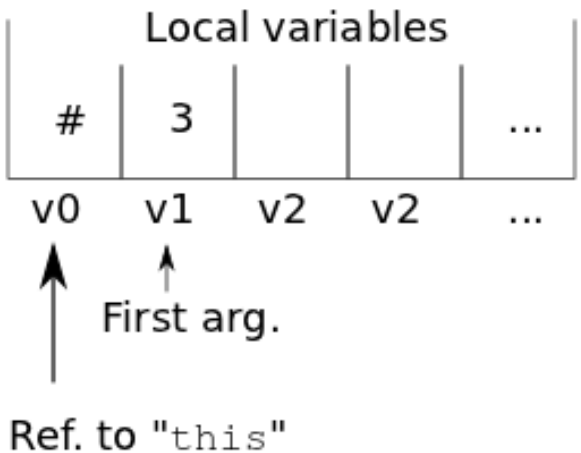
```
0:  iconst_1  // load constant 1 into stack
1:  istore_2  // store top stack in var 2
2:  iload_1   // load from var 1 to stack
3:  iload_2   // load from var 2 to stack
4:  iadd      // add 2 values on top of stack
5:  ireturn
```

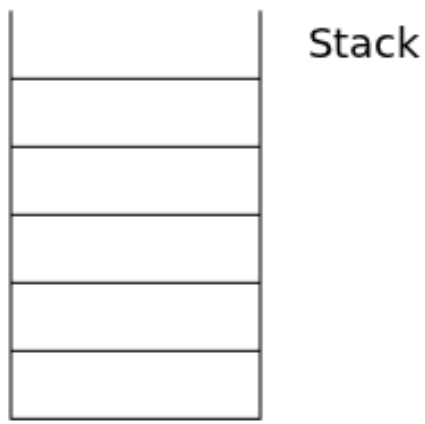
Java bytecode generation + execution overview

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

Execution of `plus(3)`:

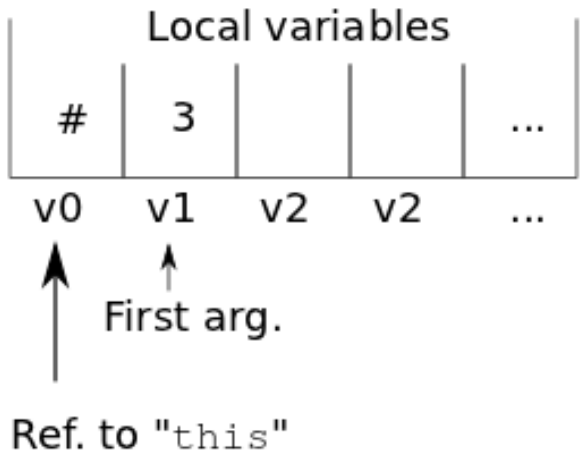


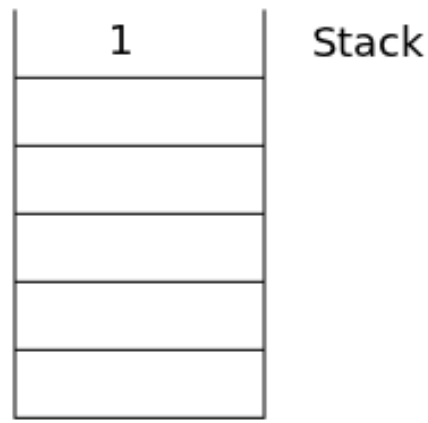


Java bytecode execution - 2

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

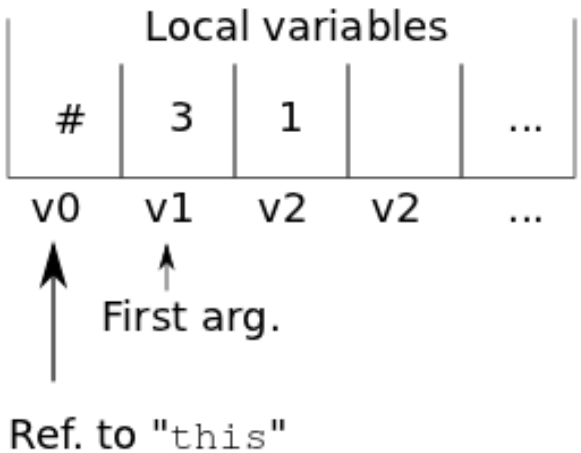


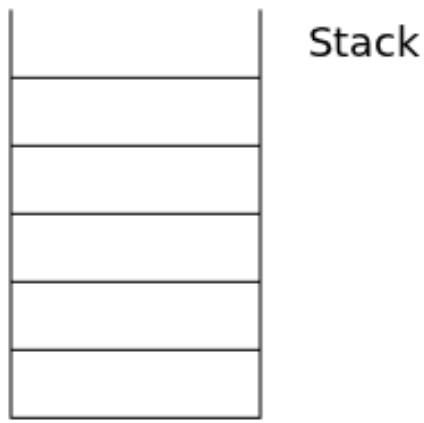


Java bytecode execution - 3

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

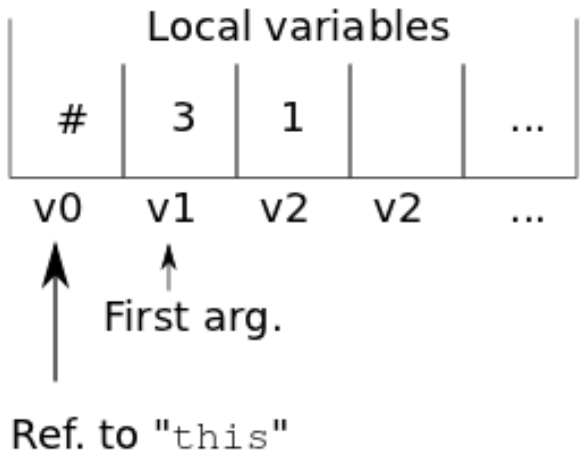


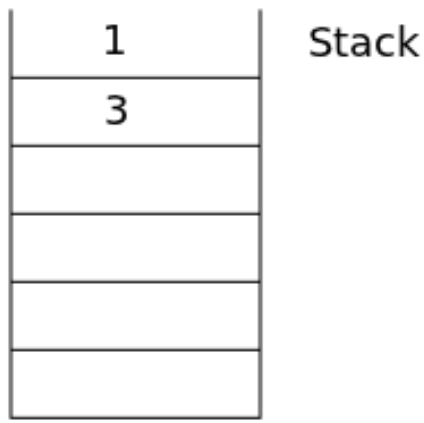


Java bytecode execution - 4

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```

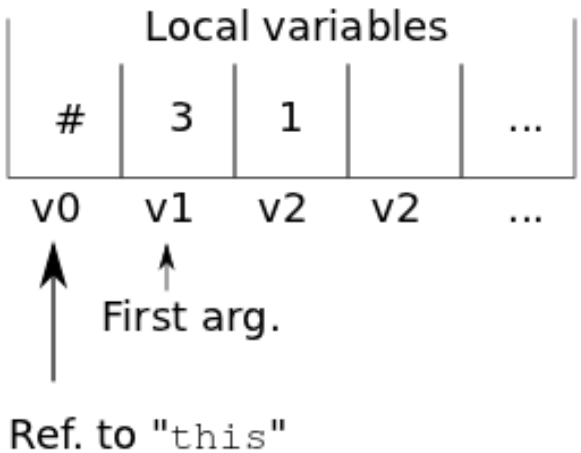


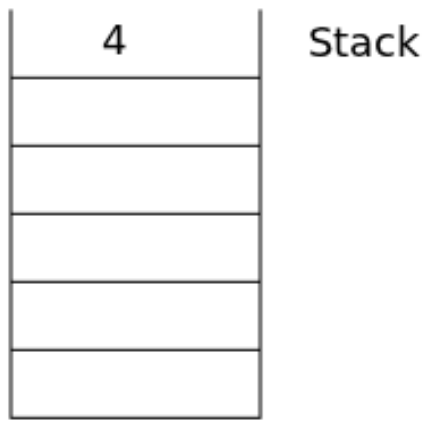


Java bytecode execution - 5

```
int plus(int a)
  {
    int b = 1;
    return a+b;
  }
```

```
0:  iconst_1   // load constant 1 into stack
1:  istore_2   // store top stack in var 2
2:  iload_1    // load from var 1 to stack
3:  iload_2    // load from var 2 to stack
4:  iadd       // add 2 values on top of stack
5:  ireturn
```





Additional bytecode considerations

- Each method has an array of local variables and a "local" stack: this is called a *frame*.
- Each *thread* has a stack of frames.
- Each class contains a *constant pool*

Example:

```
$ javap -c -s -verbose Rand
```

From bytecode to program states

From `Rand.java`:

```
[...]
int a = random.nextInt(2);
i= 1;
int b = random.nextInt(3);
[...]
```

```
14: iconst_2
15: invokevirtual #6
     // java/util/Random.nextInt:(I)
18: istore_3
19: iconst_1
20: istore_1
21: aload_2
22: iconst_3
23: invokevirtual #6
     // java/util/Random.nextInt:(I)
26: istore          4
```

Line 15 and 23 return *non-deterministic* values.

Choice generators and JPF states

- JPF creates a choice whenever multiple execution paths can arise (non-deterministic choices, user input, thread scheduling).
- **The byte-code comprised between two choices defines a JPF state.**
- JPF can store and explore states using various search strategies.

Additional JPF features

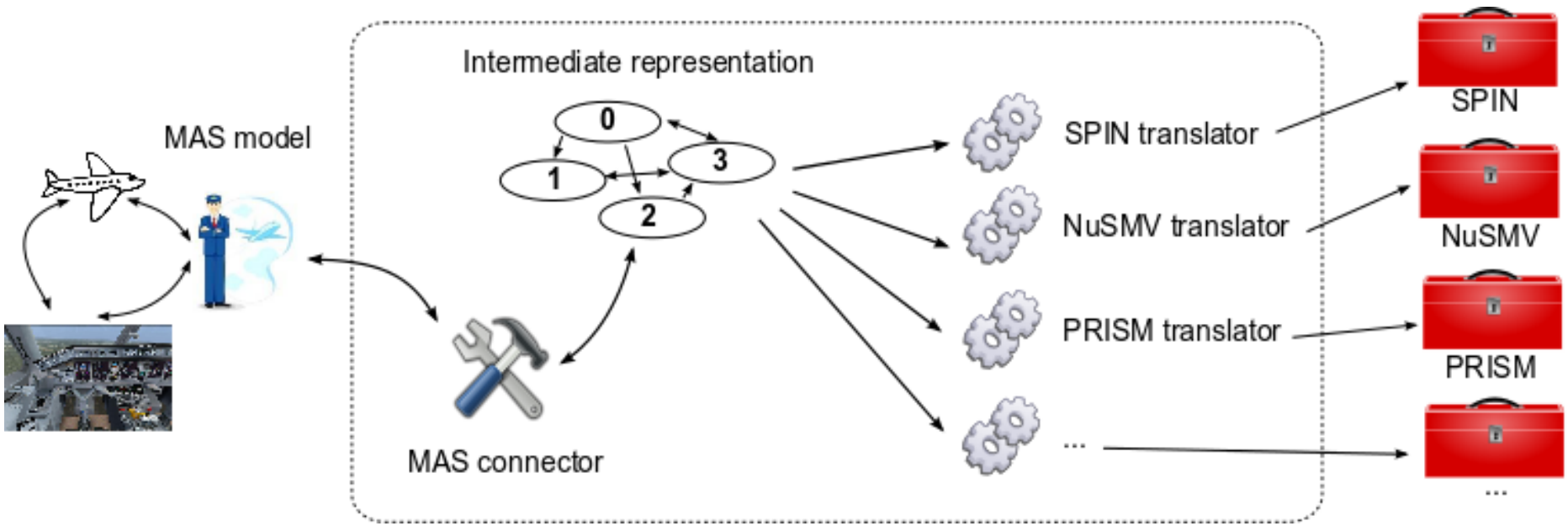
- It is possible to write custom choice generators.
- It is possible to add *listeners*: for new states, but also for specific bytecode instructions.
- It is possible to write custom state matching mechanisms.
- It is possible to write custom search strategies (e.g.: DDFS for LTL verification).

(end of JPF tutorial)

NOTICE: I'm **not** suggesting that we should use JPF for MAS verification! But it can help...

The role of JPF in MAS verification

Build a bridge between the "real" system and the model checkers for MAS.



The basic idea

1. The MAS model is what a developer produces (e.g.: a Brahms model), together with its simulation / execution environment.
2. The *intermediate representation* encodes the set of *reachable states* and the transitions. It could be explicit state, or symbolic.
3. The *connector* is used to "inspect" and "drive" the behaviour of the MAS.
4. *Translators* can be developed from the intermediate representation to the input language of existing tools.

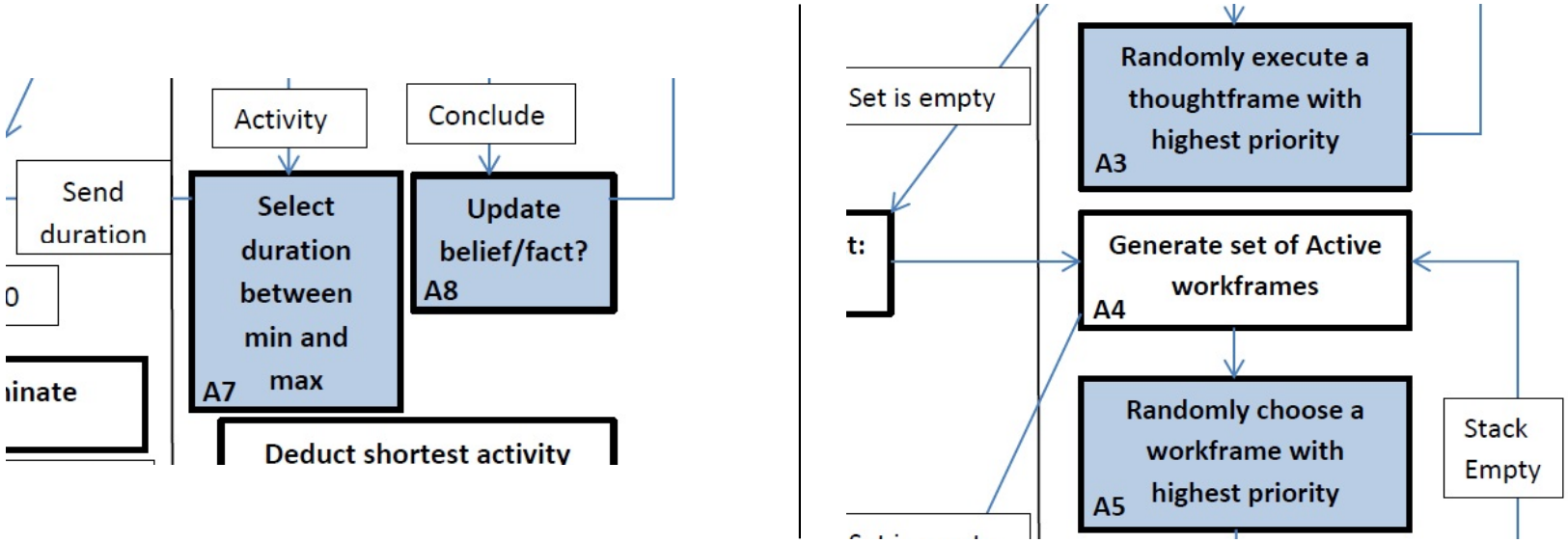
A concrete instance

1. We used Brahms as the modelling language
2. We used JPF as a connector
3. We used an explicit-state representation (a simple Java Set!)
4. We built translators to SPIN, NuSMV, and PRISM.

Brahms

- Brahms is a development and simulation environment.
- Used to model humans, robots, automated systems, agents, and interactions between humans and automated systems.
- Brahms has similarities to BDI architectures
- A Brahms model contains a set of Objects and Agents. Each of these has attributes, activities, beliefs, facts, workframes, thoughtframes etc. Syntax *very* similar to Java.
- Formal operational semantics have been defined. A scheduler is used to *simulate* possible executions.

Non-determinism in the simulator



In the corresponding Java implementation there are non-deterministic choices. For A8:

```
public boolean update (int certainty) {
  [...]
  int random = rgen.nextInt(99);
  [...]
}
```

Application 1: temporal properties of AF 447

On June 1, 2009 the Air France Flight 447 between Rio de Janeiro and Paris crashed in the equatorial Atlantic The inexperience of the pilot was determined to be the cause of the crash. The pilot in charge misjudged the airspeed of the plane (because of failure of Pitot tubes) and increased the altitude of the plane without realizing the plane was in a stall which eventually led to its crash. According to the report the pilot was presented with several chances to recover, but, was unable to do so.
Brahms model created in conjunction with aviation safety experts to show that the pilot could always correct the stall in a timely manner and that the plane does not crash due to hardware failures.
Here: **28,648 reachable states** generated in **2.5 minutes** by JPF and verification with SPIN in less than 2 sec.

Application 2: Situational Awareness for AF 447

Same scenario, but situational awareness of pilot expressed as a (temporal-) epistemic properties:

$$EF(\mathrm{actualStall} \wedge B^{Pilot}_{<0.05} actualStall)$$

In this case, state space generated by JPF and verification performed on directly on the intermediate representation.

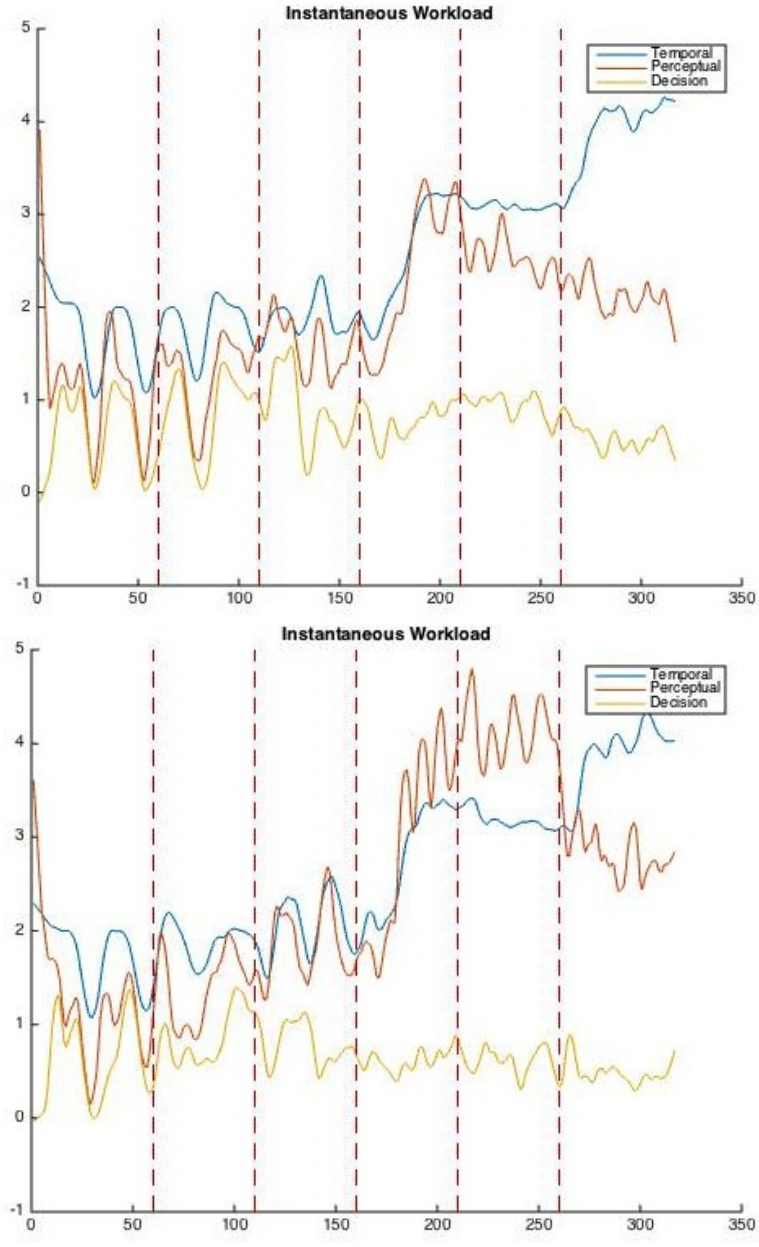
Application 3: Workload Assessment

Two Brahms scenarios:

1. Driver distracted while driving (phone call at road crossing).
2. From two pilots to single pilot operation for commercial flights.

JPF used to intercept "events" that increase workload.

Application 3: Workload Assessment



Conclusion

- In my experience: existing tools are good if "starting from scratch".
- But it is difficult to translate / encode existing scenarios.
- Moreover, this translation could be inefficient.
- Final users have very specific needs, maybe just one formula. They may use tools in ways we didn't think of, making a small extension to achieve their goals.
- JPF allows moving model checking "closer" to MAS.
- My suggestion: provide APIs, release open source, provide examples and tutorials, so that verification becomes a *chain* of techniques and tools (JPF is just one possible link).

Thank you!