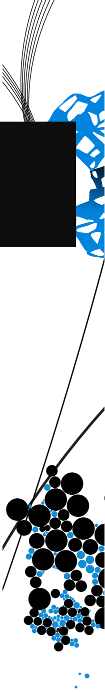
Advances and Challenges in Model-based Deductive Verification of Programs

Robert[1] Rubbens

June 16th, 2023

UNIVERSITY OF TWENTE.

Formal Methods & Tools



[1] (Bob)

Overview

1. Model-based verification
2. Model-based verification with VerCors & JavaBIP
3. VerCors Process Models
   - How to apply?

Model-based verification

No verification[2]

Listing 1: Your average average function

```c
void average(int a, int b) {
  return (a + b) / 2;
}
```

Correct?

[2] https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223

No verification[2]

Listing 1: Your average average function

```
void average(int a, int b) {
  return (a + b) / 2;
}
```

Correct? No, overflow

[2] https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223

No verification[4]

How about this patented[3] version:

```
int average(int a, int b)
{
    return (a / 2) + (b / 2) + (a & b & 1);
}
```

Correct?

[3] https://patents.google.com/patent/US6007232A/en

[4] https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223

No verification[4]

How about this patented[3] version:

```
int average(int a, int b)
{
    return (a / 2) + (b / 2) + (a & b & 1);
}
```

Correct? Maybe! Java “doesn’t” have `unsigned`!

[3] https://patents.google.com/patent/US6007232A/en

[4] https://devblogs.microsoft.com/oldnewthing/20220207-00/?p=106223

“Plain” verification

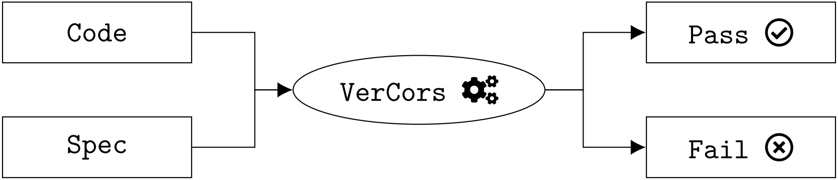
Now with contract:

```
//@ ensures \result == (a.asInt + b.asInt) / 2;
int average(int a, int b)
{
    return (a / 2) + (b / 2) + (a & b & 1);
}
```

Correct? If VerCors says so!

VerCors

- Auto-active deductive verifier
- Supports concurrent Java, (little bit of) C/OpenCL/CUDA, PVL
- Contract specifications: pre- and postconditions



Code

Spec

VerCors

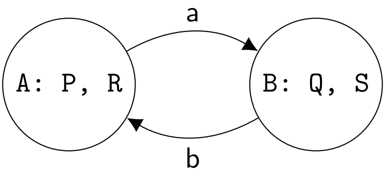
Pass ✓

Fail ⊗

Friction in plain verification

```
//@ requires state == STATE_A ==> P;
//@ requires state == STATE_B ==> Q;
//@ ensures state == STATE_A ==> R;
//@ ensures state == STATE_B ==> S;
int fooTheBar() {
    // ... implementation ...
}
```

Model-based verification



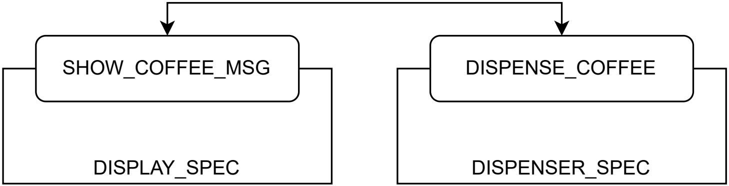
```
//@ requires state == A || state == B;
//@ ensures state == A || state == B;
int fooTheBar() {
    // ... implementation ...
}
```

Model-based verification with VerCors & JavaBIP

JavaBIP component

```
@Component(initial=IDLE, name=DISPLAY_SPEC)
class CoffeeMachineDisplay {
  int displayPort, status;
  @Transition(
    name=SHOW_COFFEE_MSG,
    source=IDLE,
    target=SHOW_PROGRESS)
  void showCoffeeMessage() {
    ...
  }
}
```

JavaBIP model



Verified JavaBIP component

```java
@Component(initial=IDLE, name=DISPLAY_SPEC)
class CoffeeMachineDisplay {
  int displayPort, status;
  @Transition(
    name=SHOW_COFFEE_MSG,
    source=IDLE,
    target=SHOW_PROGRESS,
    requires="displayPort != 0",
    ensures="status == ON")
  void showCoffeeMessage() {
    ...
  }
}
```

Our contribution

- Contracts facilitate combination of JavaBIP with VerCors
- Verify JavaBIP models deductively
- Check contracts at runtime
- Optimize away runtime checks
- Casino case study to illustrate tool

Paper: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java
DOI: `10.1007/978-3-031-30826-0_8`

VerCors Process Models

VerCors Process Models

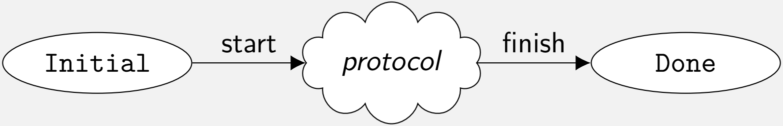
- Developed during Wytse Oortwijn's PhD thesis[5]
- Used to verify leader election protocol
- Goal/"cool feature":
  - Establish property in model
  - With VerCors, check link between model and program deductively
  - Assume properties of model in program

[5] `https://doi.org/10.3990/1.9789036548984`

Example: leader election

Process model

ensures $\forall n \in nodes.\ n.leader = maxNode(nodes)$;
action finish();



Example: leader election

Process model

```
ensures $\forall n \in nodes.\ n.leader = maxNode(nodes)$;
action finish();
```

Example: leader election

Process model

```
ensures ∀n ∈ nodes. n.leader = maxNode(nodes);
action finish();
```

Code + spec

```
//@ requires m.Initial();
int electLeader(Model m) {
    protocol(m);
    action M.finish();
    assert M.Done();
    assert ∀n ∈ nodes. n.leader = maxNode(nodes);
}
```

Limitations

- As currently defined, not clear how to parameterize
    - No backend that can handle this

Limitations

- As currently defined, not clear how to parameterize
  - No backend that can handle this
- Process-algebraic specification might be too abstract
  - What about other forms? Session types? Imperative specifications?
  - Do process-algebraic specs scale?

Limitations

- As currently defined, not clear how to parameterize
  - No backend that can handle this
- Process-algebraic specification might be too abstract
  - What about other forms? Session types? Imperative specifications?
  - Do process-algebraic specs scale?
- Designed with model checker in mind
  - LTL/CTL/mu calculus is powerful
  - But: how to “assume” and LTL formula in a deductive setting?

Where to apply?

- The usual way:
  1. Find a bigger toy example
  2. Look for interesting properties
  3. `goto` 1, until a large case study appears

Where to apply?

- The usual way:
  1. Find a bigger toy example
  2. Look for interesting properties
  3. `goto` 1, until a large case study appears
- Shortcut: ideas from practice, industry
  - Schedulers?
  - Protocols?
  - Threads that work together for some concrete goal?

Conclusion

- Model-based verification seems a logical step
- Current formulation seems effective, but difficult to apply
- Where to go next?
  - Resolve limits?
  - Look harder for case studies?

Conclusion

- Model-based verification seems a logical step
- Current formulation seems effective, but difficult to apply
- Where to go next?
  - Resolve limits?
  - Look harder for case studies?
- Come talk to me afterwards!
  - Ideas for possible applications
  - To tell me I’m wrong `:D`

Paper: JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java
DOI: `10.1007/978-3-031-30826-0_8`

Robert Rubbens
Formal Methods & Tools, University of Twente
r.b.rubbens@utwente.nl

Bonus slides

Verified JavaBIP: implementation

- In VerCors:
  1. Parse Verified JavaBIP annotations
  2. Encode contracts using JavaBIP semantics into `COL`
  3. Verify `COL` program
  4. Translate back any errors to input
  5. Produce verification report
- In the JavaBIP engine:
  1. Parse Verified JavaBIP annotations
  2. If supplied, import verification report
  3. Runtime verification
     - Check non-verified properties at points of interest