UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

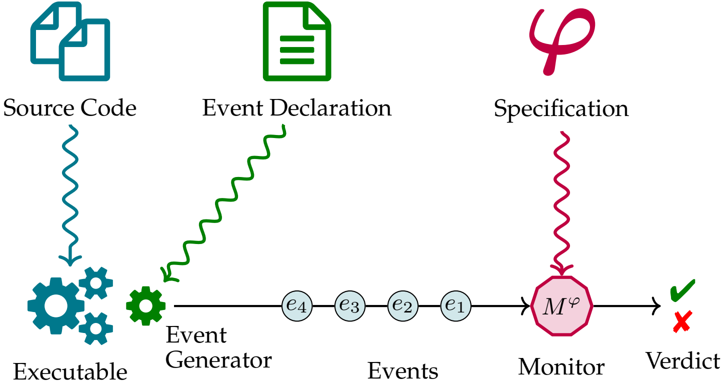
Optimizing Trans-Compilers in Runtime Verification makes Sense – Sometimes

Hannes Kallwies Martin Leucker Meiko Prilop Malte Schmitz

Institute for Software Engineering and Programming Languages,
University of Lübeck, Lübeck, Germany

16th International Symposium on Theoretical Aspects of Software Engineering, July 2022

Runtime Verification: General idea



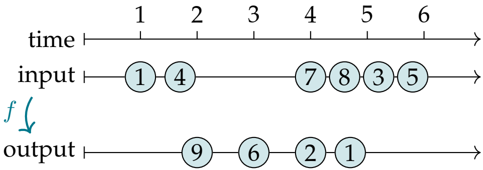
Stream Runtime Verification

Idea: Use dataflow-oriented languages (similar to Lustre, Lucid, Esterell...) to describe system properties and generate monitors.

Basic concept: Input streams are combined with operators to generate output streams.

Popular SRV languages:

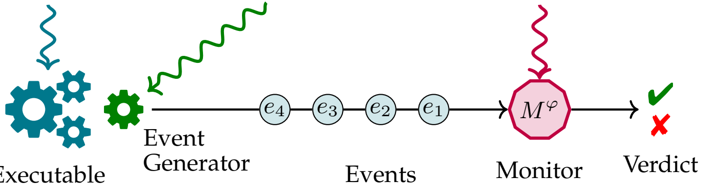
- LOLA
- Striver
- **TeSSLa**



Stream Runtime Verification with TeSSLa

```c
int main() {
  while (1) {
    lock();
    critical();
    unlock();
  }
}
```

```
@InstFunctionCall("lock")
    in lock: Events[Unit]
@InstFunctionCall("unlock")
    in unlock: Events[Unit]
@InstFunctionCall("critical")
    in crit: Events[Unit]

    out on(crit, count(lock) -
                 count(unlock) == 1)
    as verdict
```



Monitor generation from a TeSSLa specification

- TeSSLa specifications can be compiled to Scala monitors (Trans-Compiler)

Monitor generation from a TeSSLa specification

- TeSSLa specifications can be compiled to Scala monitors (Trans-Compiler)
- Every TeSSLa specification can be described by 6 core stream operators + function definitions (TeSSLa Core)

Monitor generation from a TeSSLa specification

- TeSSLa specifications can be compiled to Scala monitors (Trans-Compiler)
- Every TeSSLa specification can be described by 6 core stream operators + function definitions (TeSSLa Core)
- TeSSLa offers possibility to define own stream operators as macros and offers Stdlib of common functions

Monitor generation from a TeSSLa specification

TeSSLa specification

```
in x: Events[Int]
def o = count(x)
```

TeSSLa Core (schema)

```
def v0: Int = 1
def setDefault = [...]
def addOne = (i: strict Option[Int]) => {
  def v1: Int = getSome(i)
  def v2: Int = add(v1, v0)
  def v3: Option[Int] = Some(v2)
  v3
}

in x: Events[Int]
def v4: Events[Int] = last(o, x)
def v5: Events[Int] = lift(v4, addOne)
def v6: Events[Int] = unit
def o: Events[Int] = lift(v5, v6, setDefault)
```

Monitor generation from a TeSSLa specification

Generated Scala Code (schema)

```
//Var declarations for non−stream constants
var v0: Int = 0
var setDefault: (Option[Int], Option[()]) => Option[Int] = null
var addOne: (Option[Int]) => Option[Int] = null

//Initialization of non−stream constants
v0 = 1
setDefault = [...]
addOne = [...]

//Variables for the state of each stream (5 streams)
var x_changed: Bool = false
var x_hasLast: Bool = false
var x_curr: Int = 0
var x_last: Int = 0
[...]

//Evaluation function
def calculate(ts: Int) = {

  //def v4: Events[Int] = last(o, x)
  if (x_changed && o_hasLast) {
    v4_last = v4_curr
    v4_curr = o_last
    [...]
  }
  [...]

}
```

Monitor generation from a TeSSLa specification

TeSSLa compiler uses straight-forward translation strategy

Monitor generation from a TeSSLa specification

TeSSLa compiler uses straight-forward translation strategy

Advantages:

- Easy to implement, no complex analysis phases
- Translation schemes required only six core operators + some basic functions

Monitor generation from a TeSSLa specification

TeSSLa compiler uses straight-forward translation strategy

Advantages:

- Easy to implement, no complex analysis phases
- Translation schemes required only six core operators + some basic functions

Disadvantages:

- Lengthy and hard to read code
- Inefficient code (lots of variable declarations, complex code)

Monitor generation from a TeSSLa specification

TeSSLa compiler uses straight-forward translation strategy

Advantages:

- Easy to implement, no complex analysis phases
- Translation schemes required only six core operators + some basic functions

Disadvantages:

- Lengthy and hard to read code
- Inefficient code (lots of variable declarations, complex code)

$\Rightarrow$ No problem, target compiler will optimize code.

Monitor generation from a TeSSLa specification

TeSSLa compiler uses straight-forward translation strategy

Advantages:

- Easy to implement, no complex analysis phases
- Translation schemes required only six core operators + some basic functions

Disadvantages:

- Lengthy and hard to read code
- Inefficient code (lots of variable declarations, complex code)

$\Rightarrow$ No problem, target compiler will optimize code. **Won't it?**

Compiler optimizations

1. Extend set of core operators by stdlib functions (**count**, **const**, **default**, **fold** ...)

Compiler optimizations

1. Extend set of core operators by stdlib functions (**count**, **const**, **default**, **fold** ...)

   ⇒ Scala-DSL in compiler to add translation schemes easily in a generic fashion.

Compiler optimizations

1. Extend set of core operators by stdlib functions (**count**, **const**, **default**, **fold** ...)

   ⇒ Scala-DSL in compiler to add translation schemes easily in a generic fashion.

2. Avoidance of dummy initialization and usage of **val** keyword where possible

Compiler optimizations

1. Extend set of core operators by stdlib functions (**count**, **const**, **default**, **fold** ...)

   ⇒ Scala-DSL in compiler to add translation schemes easily in a generic fashion.

2. Avoidance of dummy initialization and usage of **val** keyword where possible

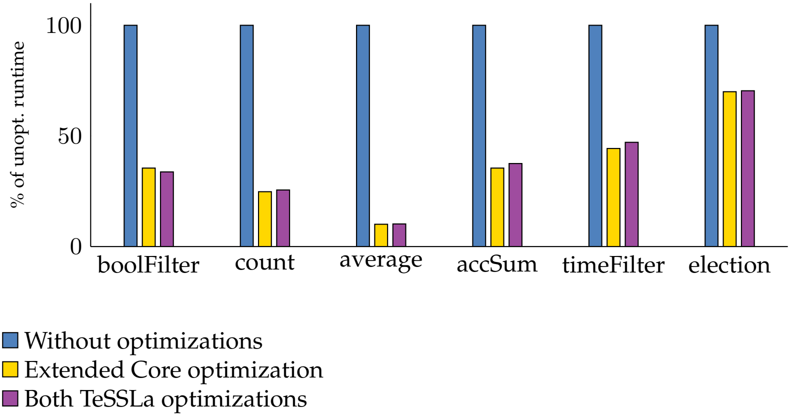
   ⇒ Simple loop analysis to detect dependency cycles.

Compiler optimizations

Generated Scala Code with optimization (schema)

```
//Var declarations for non−stream constants
val v0: Int = 0
val setDefault: (Option[Int], Option[()]) => Option[Int] = [...]
val addOne: (Option[Int]) => Option[Int] = [...]

//Variables for the state of each stream (2 streams)
var x_changed: Bool = false
var x_hasLast: Bool = false
var x_curr: Int = 0
var x_last: Int = 0
[...]

//Evaluation function
def calculate(ts: Int) = {

  //def o: Events[Int] = count(x)
  if (x_changed) {
    o_last = o_curr
    o_curr = o_curr + 1
    o_changed = true
    o_hasLast = true
  }

}
```

Evaluation



100
50
0
% of unopt. runtime
boolFilter
count
average
accSum
timeFilter
election
Without optimizations
Extended Core optimization
Both TeSSLa optimizations

Conclusion

- For monitor generation TeSSLa is compiled to TeSSLa Core and then to Scala code
- Straight-forward strategy produces inefficient code
- Introducing new core operators led to higher monitor performance
- Avoiding dummy initialization did not to affect the monitor performance

Conclusion

- For monitor generation TeSSLa is compiled to TeSSLa Core and then to Scala code
- Straight-forward strategy produces inefficient code
- Introducing new core operators led to higher monitor performance
- Avoiding dummy initialization did not to affect the monitor performance

Insight: While small optimizations do often not lead to performance gains in trans-compilers, optimizations affecting significantly the structure of the code may improve the overall performance.

Conclusion

- For monitor generation TeSSLa is compiled to TeSSLa Core and then to Scala code
- Straight-forward strategy produces inefficient code
- Introducing new core operators led to higher monitor performance
- Avoiding dummy initialization did not to affect the monitor performance

Insight: While small optimizations do often not lead to performance gains in trans-compilers, optimizations affecting significantly the structure of the code may improve the overall performance.

$\Rightarrow$ **Optimizing Trans-Compilers in Runtime Verification makes Sense – Sometimes**