Formal Methods for Space Electronics

Georgy Lukyanov, supervised by Andrey Mokhov, Alexander Romanovsky, Jakob Lechner

School of Engineering · School of Engineering · School of CS · RUAG Space Austria

Motivation

Verification of functional correctness of control programs is an essential task for the development of space electronics; it is difficult and time-consuming and typically outweighs design and programming tasks in terms of development hours. We present a verification approach designed to help engineers reduce the effort required for formal program verification.

The approach uses a ***metalanguage*** to describe the semantics of a program as a ***state transformer***, which can be compiled to multiple targets for the purpose of verification and code generation.

Haskell

```
energyEstimate :: (Integral a, Num a)
  => a -> a  -> a -> a -> a
energyEstimate t1 t2 p1 p2 =
  abs (t1 - t2) * (p1 + p2) `div` 2
```

Typed Expression

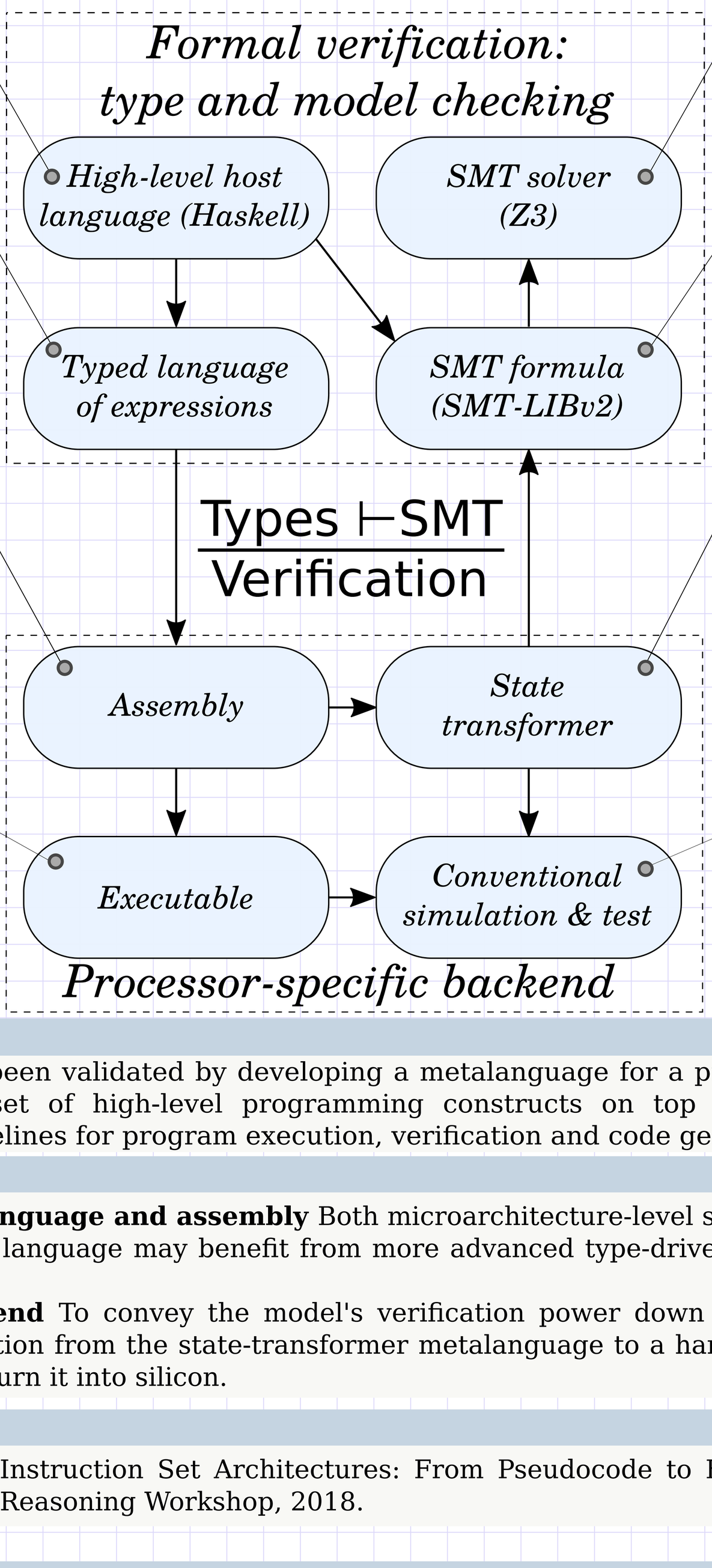
```
energyEstimateHighLevel = do
    let t1 = read (IntegerVariable 0)
        t2 = read (IntegerVariable 1)
        p1 = read (IntegerVariable 2)
        p2 = read (IntegerVariable 3)
        temp = Temporary 4
        s = Stack 5
    compile r0 s temp
            (energyEstimate t1 t2 p1 p2)
    halt
```

Assembly

```
energyEstimateLowLevel = do
    ld r0 0
    sub r0 1
    abs r0
    ld r1 2
    add r1 3
    st r1 3
    mul r0 3
    div_si r0 2
    halt
```

Binary Code

```
0010000000000000
0011001111111000
1001011111111000
0100111111111000
1000101111111000
0000000000000000
```



Verification Result

```
> proveWith z3 $ theorem
Falsifiable. Counter-example:
  t1 = 5190405167614263295 :: Int64
  t2 =                   0 :: Int64
  p1 =  149927859193384455 :: Int64
  p2 =  157447350457463356 :: Int64
```

Constraints

```
p1 <- forall "p1"
p2 <- forall "p2"
constrain $ p1 .>= 0 &&& p1 .<= 1000
constrain $ p2 .>= 0 &&& p2 .<= 1000
let result =
  readArray (registers finalState) 0
let halted =
  readArray (flags finalState) (flagId Halt)
pure $   halted
     &&& result .>= 0
```

State Transformer

```
data Machine a = Machine {
  machine :: State -> (a, State) }

jmpi_ct :: SymbolicValue Int8 -> Machine ()
jmpi_ct offset = do
  ic <- readInstructionCounter
  condition <- readFlag Condition
  let ic' =
    ite condition (ic + offset) ic
  writeInstructionCounter ic'
```

Simulation Output

```
Memory Dump: [ ( 0, 10 )  R0: 20
             , ( 1, 5  )
             , ( 2, 3  )
             , ( 3, 5  )
             , ( 4, 5  )
             , ( 5, 100) ]
```

Conclusion

The presented approach has been validated by developing a metalanguage for a processing core designed for space missions, defining a set of high-level programming constructs on top of the metalanguage, and implementing compilation pipelines for program execution, verification and code generation.

Future Work

- **Dependently-typed metalanguage and assembly** Both microarchitecture-level semantics and instruction set architecture level modelling language may benefit from more advanced type-driven verification via devendent types.
- **Hardware synthesis backend** To convey the model's verification power down to the bare metal, we may implement a verified translation from the state-transformer metalanguage to a hardware description language to petrifythe semantics and turn it into silicon.

Related Work

- **SAIL** "Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics." Alasdair Armstrong at al., Automated Reasoning Workshop, 2018.

Newcastle University

g.lukyanov2@newcastle.ac.uk

geo2a.info

Together ahead. RUAG