Formal Methods for Space Electronics

Formal Verification of Spacecraft Control Programs Using a Metalanguage for State Transformers

Georgy Lukyanov
Newcastle University

Supervisors: **Andrey Mokhov**[1], **Alexander Romanovsky**[1], **Jakob Lechner**[2]
[1]Newcastle University, [2]RUAG Space Austria

{g.lukyanov2@newcastle.ac.uk}
March 23, 2018

Newcastle University

Together ahead. RUAG

Agenda

1. REDFIN instruction set architecture
2. REDFIN formal model
3. Verification example
4. Conclusion

REDFIN ISA

REDuced instruction set for Fixed-point & INteger arithmetic

REDFIN: minimalistic sequencer for space missions

Goals

- Simple instruction set to achieve a small hardware footprint
- Reduced complexity to support formal verification of programs
- Deterministic behaviour for real-time applications

Facts

- Configurable bit width for the data path, ranging from 8 to 64 bits
- 47 instructions
- 4 general purpose registers
- No caches, no pipelining, no speculative execution
- Realised with a space-qualified FPGA
- Deployed as part of an antenna pointing unit for satellites

Formal Model & Verification Framework

REDFIN verification framework: features

- Haskell-embedded assembly language
- A "compiler" from a subset of Haskell (arithmetics) to REDFIN assembly
- Microarchitecture state transformer semantics (in Haskell)
- Symbolic execution of programs
  (via SBV https://hackage.haskell.org/package/sbv)
- Verification of user-defined program properties via SMT solver (Z3)

Properties to verify

- Status of a certain flag (`Overflow`, `Halt`, `OutOfMemory` etc.)
- Threshold on the execution time (amount of system clock cycles)
- Correctness of the computed result
- Equivalence of programs (in terms of output)

State &
State Transformers*

* No politics or electrical grids involved

Definition 1. REDFIN microarchitecture state

$$S = \{(r, m, ic, ir, p, f, c) : r \in R, m \in M, ic \in A, ir \in I, p \in P, f \in F, c \in C\}$$

```
data State = State { registers           :: RegisterBank
                   , memory              :: Memory
                   , instructionCounter  :: InstructionAddress
                   , instructionRegister :: InstructionCode
                   , program             :: Program
                   , flags               :: Flags
                   , clock               :: Clock }
```

Definition 1. REDFIN microarchitecture state

$$S = \{(r, m, ic, ir, p, f, c) : r \in R, m \in M, ic \in A, ir \in I, p \in P, f \in F, c \in C\}$$

```
data State = State { registers          :: RegisterBank
                   , memory             :: Memory
                   , instructionCounter  :: InstructionAddress
                   , instructionRegister :: InstructionCode
                   , program            :: Program
                   , flags              :: Flags
                   , clock              :: Clock }
```

Definition 2. State transformer

$$\tau : S \to S$$

A function mapping states to states.

Definition 1. REDFIN microarchitecture state

$$S = \{(r, m, ic, ir, p, f, c) : r \in R, m \in M, ic \in A, ir \in I, p \in P, f \in F, c \in C\}$$

```
data State = State { registers           :: RegisterBank
                   , memory              :: Memory
                   , instructionCounter  :: InstructionAddress
                   , instructionRegister :: InstructionCode
                   , program             :: Program
                   , flags               :: Flags
                   , clock               :: Clock }
```

Definition 2. State transformer

$$\tau : S \to S$$

A function mapping states to states.

```
data Redfin a = Redfin { transform :: State -> (a, State) }

transformState :: (State -> State) -> Redfin ()
transformState f = Redfin $ \s -> ((), f s)
```

Definition 1. REDFIN microarchitecture state

$$S = \{(r, m, ic, ir, p, f, c) : r \in R, m \in M, ic \in A, ir \in I, p \in P, f \in F, c \in C\}$$

```
data State = State { registers           :: RegisterBank
                   , memory              :: Memory
                   , instructionCounter  :: InstructionAddress
                   , instructionRegister :: InstructionCode
                   , program             :: Program
                   , flags               :: Flags
                   , clock               :: Clock }
```

Definition 2. State transformer

$$\tau : S \to S$$

A function mapping states to states.

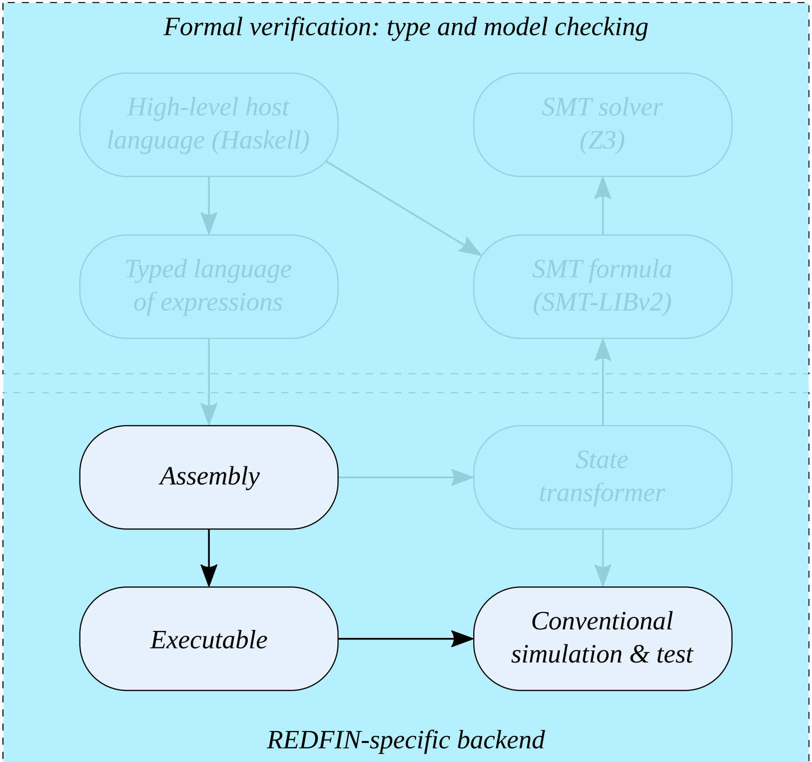
```
data Redfin a = Redfin { transform :: State -> (a, State) }

transformState :: (State -> State) -> Redfin ()
transformState f = Redfin $ \s -> ((), f s)
```

Example: system clock advancement

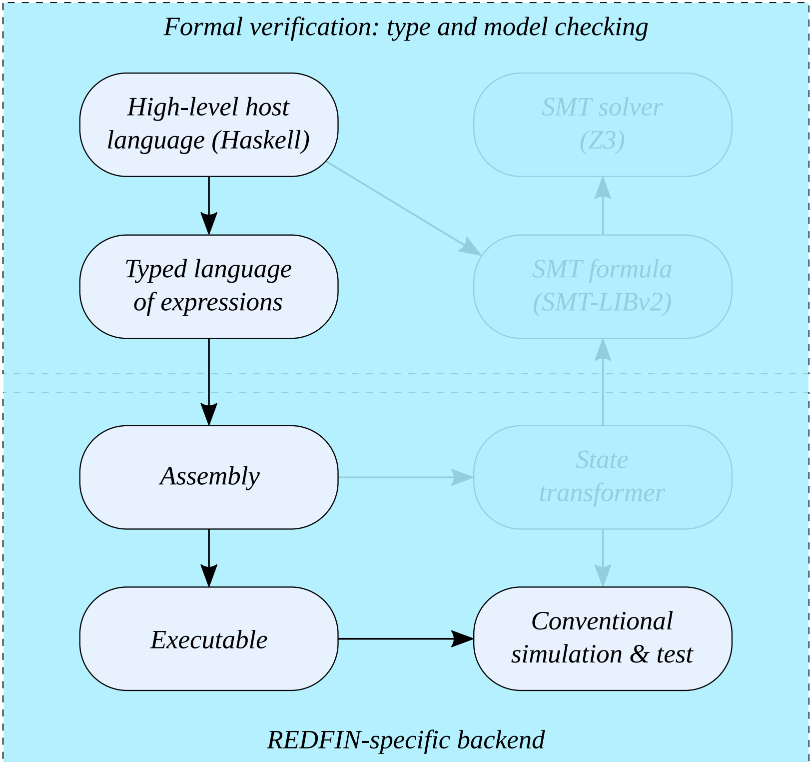
```
delay cycles = transformState $ \(State rs ic ir fs m p  c ->
                                  State rs ic ir fs m p (c + cycles)
```

Workflow options

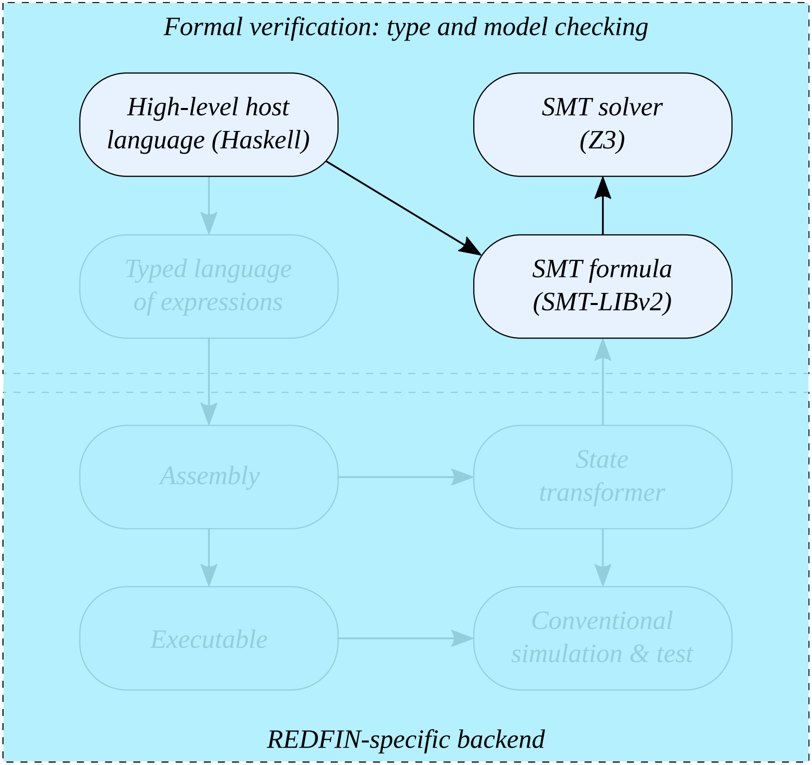
Conventional workflow: non-verified assembly programming



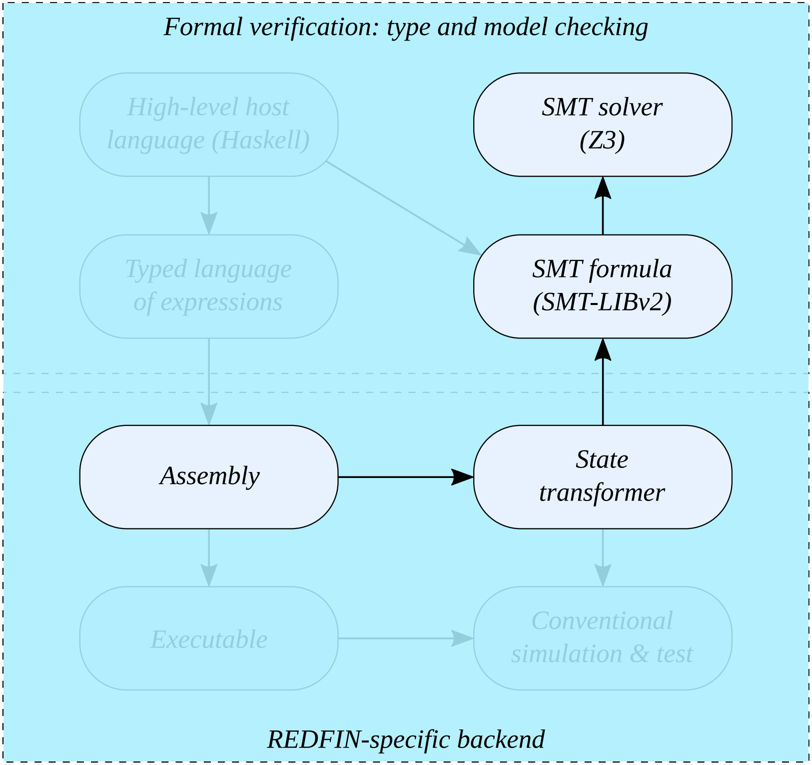
Deriving assembly programs from Haskell



Verifying Haskell programs

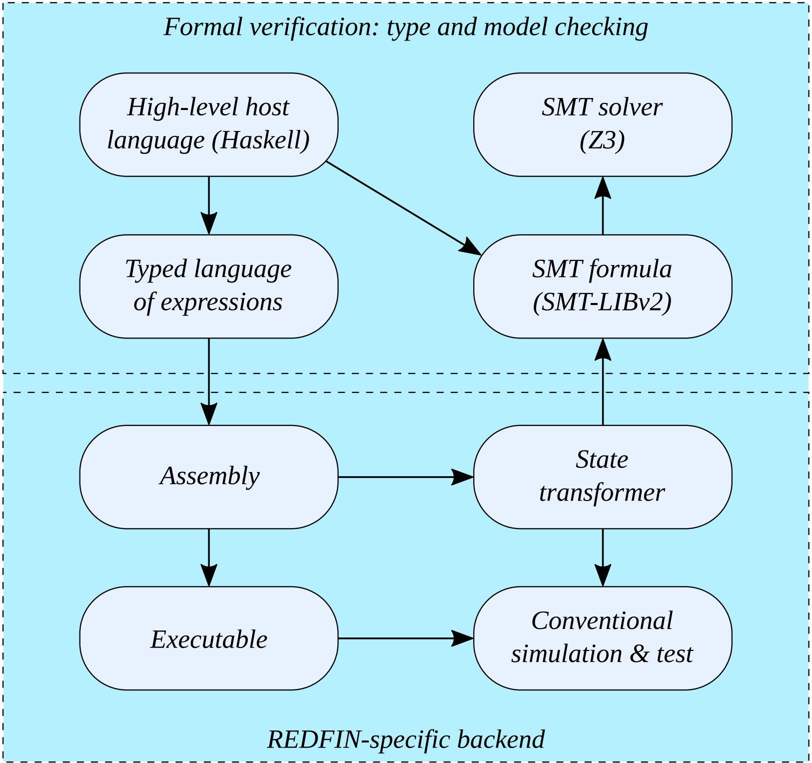


Verifying assembly programs



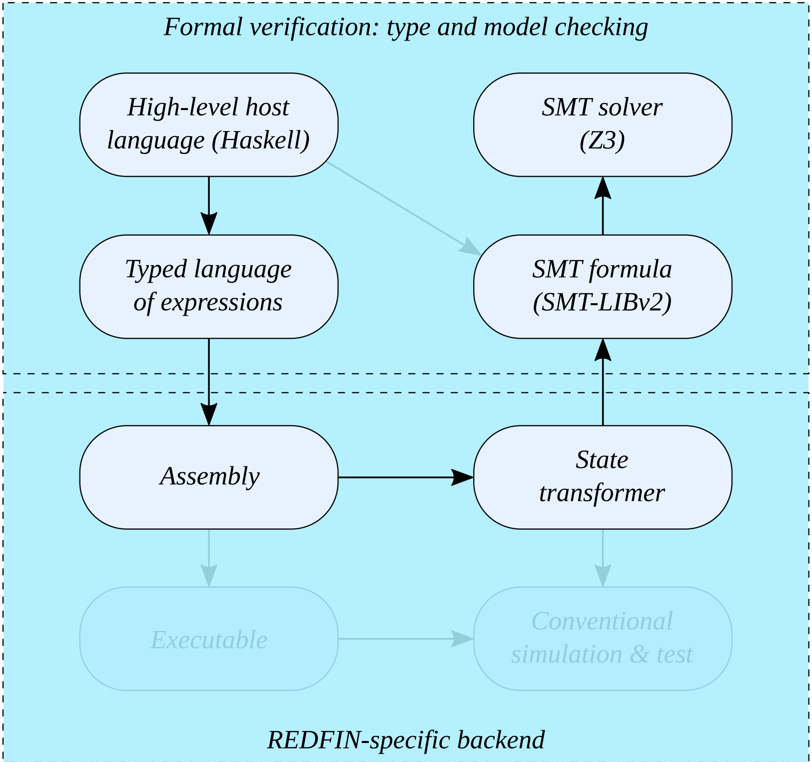
One Workflow to rule them all

Deriving assembly from Haskell and verifying equivalence



Example

Verifying arithmetics



Haskell arithmetical expression

```
addHaskell :: Integral a => a -> a -> a
addHaskell x y = x + y
```

Haskell

Embedding Haskell to REDFIN assembly

```
addHighLevel = do
    let x     = read $ IntegerVariable 0
        y     = read $ IntegerVariable 1
        temp  = Temporary 3
        stack = Stack 5
    compile r0 stack temp (addHaskell x y)
    halt
```

Typed language of expressions

Compilation result (roughly)

```
addLowLevel = do
    let { x = 0; y = 1 }
    ld r0 x
    add r0 y
    halt
```

Assembly

Checking for integer overflow

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Checking for integer overflow

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Executing the SMT solver…

```
ghci> proveWith z3 addNoOveflow
...
```

Checking for integer overflow

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Executing the SMT solver... and **BOOM!**

```
ghci> proveWith z3 addNoOveflow
Elapsed time: 0.047s
Falsifiable. Counter-example:
  t1 = 8748242276167214084 :: Int64
  t2 = 8646348300372410368 :: Int64
```

Checking for integer overflow

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Executing the SMT solver… and **BOOM!**

```
ghci> proveWith z3 addNoOveflow
Elapsed time: 0.047s
Falsifiable. Counter-example:
  t1 = 8748242276167214084 :: Int64
  t2 = 8646348300372410368 :: Int64
```

```
ghci> (8748242276167214084 :: Int64) + (8646348300372410368 :: Int64)
-1052153497169927164
```

Considering input constraints

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    constrain $ x .>= 0 &&& x .<= 10 ^ 6
    constrain $ y .>= 0 &&& y .<= 10 ^ 6
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Considering input constraints

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    constrain $ x .>= 0 &&& x .<= 10 ^ 6
    constrain $ y .>= 0 &&& y .<= 10 ^ 6
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Executing the SMT solver…

```
ghci> proveWith z3 addNoOveflow
...
```

Considering input constraints

```
addNoOverflow = do
    x <- forall "t1"
    y <- forall "t2"
    constrain $ x .>= 0 &&& x .<= 10 ^ 6
    constrain $ y .>= 0 &&& y .<= 10 ^ 6
    let mem        = initialiseMemory [(0, x), (1, y), (3, 100)]
        finalState = simulate 100 $ boot addHighLevel mem
        result     = readArray (registers finalState) 0
        overflow   = readArray (flags finalState) (flagId Overflow)
    pure $ bnot overflow
```

Executing the SMT solver... All good.

```
ghci> proveWith z3 addNoOveflow
Elapsed time: 0.029s
Q.E.D.
```

Restrictions

What can be verified

- Integer and fixed-point arithmetics (absence of overflow, correctness of result, etc.)
- Bounded loops, i.e. sum of an array of a given length
- Threshold on termination time
- A lot of more useful cases

What we cannot verify

- Unbounded loops (hello Halting, my old friend)
- "Big" (sorting an array of 50 numbers) problems require a lot of time

Conclusion

REDFIN verification framework overview

- ~2000 LOC, Haskell
- High-level typed language compiled to REDFIN assembly
- Low-level Haskell-embedded assembly language
- Checking microarchitecture state properties
- Checking equivalence of programs

Extra features

- Worst-case execution time analysis
- C-code generation for massive parallel testing

REDFIN verification framework overview

- ~2000 LOC, Haskell
- High-level typed language compiled to REDFIN assembly
- Low-level Haskell-embedded assembly language
- Checking microarchitecture state properties
- Checking equivalence of programs

Extra features

- Worst-case execution time analysis
- C-code generation for massive parallel testing

Get in touch

Georgy Lukyanov {g.lukyanov2@newcastle.ac.uk}

- Tuura website: https://tuura.org/
- Github for REDFIN source code (availible soon): https://github.com/tuura
- The REDFIN paper draft (feedback wanted!): https://arxiv.org/abs/1802.01738