Formal Verification of Spacecraft Control Programs

GEORGY LUKYANOV, Newcastle University, United Kingdom
ANDREY MOKHOV, Newcastle University, United Kingdom
JAKOB LECHNER, RUAG Space GmbH, Austria

Verification of correctness of control programs is an essential task in the development of space electronics; it is difficult and typically outweighs design and programming tasks in terms of development hours. This paper presents a verification approach designed to help spacecraft engineers reduce the effort required for formal verification of low-level control programs executed on custom hardware.

The verification approach is demonstrated on an industrial case study. We present REDFIN, a processing core used in space missions, and its formal semantics expressed using the proposed metalanguage for state transformers, followed by examples of verification of simple control programs.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; • **Computer systems organization** → *Embedded software.*

Additional Key Words and Phrases: formal verification, instruction set architecture, functional programming, domain-specific languages.

ACM Reference Format:
Georgy Lukyanov, Andrey Mokhov, and Jakob Lechner. 2020. Formal Verification of Spacecraft Control Programs. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2020), 18 pages. https://doi.org/10.1145/3391900

1 INTRODUCTION

Software bugs play a major role in the history of spacecraft accidents [13]. Some of the known mission-ending bugs, e.g. due to software updates, would have been difficult to prevent, but integer overflows [3] and incorrect unit conversion [16] should have been eradicated long ago. This paper combines known formal verification and programming languages techniques and presents a formal verification approach for simple control tasks, such as satellite power management, which are executed on a real processing core used in space missions.

Fig. 1 shows an overview of our approach. The bottom part corresponds to conventional code generation and test, where REDFIN[1] assembly language is executed by simulating the effect of each instruction on the state of the processor and memory. The corresponding *state transformer* is typically implicit and intertwined with the rest of the simulation infrastructure. The main idea of our approach is to represent the state transformer explicitly so that it can be symbolically manipulated and used not only for simulation but also for formal verification. The latter is achieved

[1]REDFIN stands for 'REDuced instruction set for Fixed-point & INteger arithmetic'. This instruction set and the corresponding processing core were developed by RUAG Space Austria GmbH for space missions (see §2).

Authors' addresses: Georgy Lukyanov, Newcastle University, United Kingdom, g.lukyanov2@ncl.ac.uk; Andrey Mokhov, Newcastle University, United Kingdom, andrey.mokhov@ncl.ac.uk; Jakob Lechner, RUAG Space GmbH, Austria, jakob.lechner@gmx.net.

1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/3391900

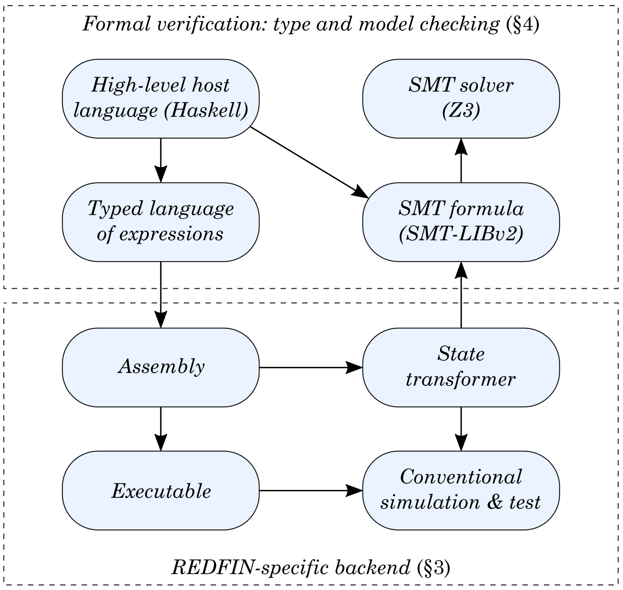
Formal verification: type and model checking (§4)
High-level host language (Haskell)
SMT solver (Z3)
Typed language of expressions
SMT formula (SMT-LIBv2)
Assembly
State transformer
Executable
Conventional simulation & test
REDFIN-specific backend (§3)

Fig. 1. Overview of the presented verification approach.

by compiling state transformers to SMT formulas and using an SMT solver, e.g. Z3 [6], to verify that certain correctness properties hold, for example, that integer overflow cannot occur regardless of input parameters and that the program always terminates within stated time.

By using Haskell as the host language we can readily implement compilers from higher-level *typed* languages to untyped assembly, eradicating incorrect number and unit conversion bugs. As shown at the top of Fig. 1, engineers can write high-level control programs for the REDFIN architecture directly in a small subset of Haskell. These high-level programs can be used for type-safe code generation and as executable specifications of intended functionality for the purposes of program synthesis and equivalence checking.

We first introduce the REDFIN processing core (§2), then present our verification approach (§3-§5), and conclude by a discussion (§6) and a review of related work (§7).

This paper is an extended version of an earlier conference paper [15]. The key changes compared to the earlier version are: (i) §3 has been expanded to describe the branching mode of symbolic simulation within the presented verification framework; (ii) an entirely new section §5 addresses verification of programs with unbounded loops on an example of a stepper motor control program; (iii) the discussion section has also been updated.

2 THE REDFIN ARCHITECTURE OVERVIEW

Many spacecraft subsystems rely on integrated circuits to perform control tasks or simple data processing. Typically, these integrated circuits are realised with Field Programmable Gate Arrays (FPGAs) benefiting from their flexibility and low cost. Modern space-qualified FPGAs that can withstand radiation in Earth orbit or deep space have a limited amount of programmable resources, and it is often not feasible to implement a fully-fledged processor system in such an FPGA next to the mission-specific circuitry. The REDFIN instruction set was developed to address this issue and meet the following goals: (i) simple instruction set with a small hardware footprint, (ii) reduced complexity to support formal verification of programs, and (iii) deterministic real-time behaviour.

2.1 REDFIN Instruction Set and Microarchitecture

REDFIN instructions have a fixed width of 16 bits. The instruction set is based on a register-memory architecture, i.e. instructions can fetch their operands from registers as well as directly from the memory. This architecture favours a small register set, which minimises the hardware footprint of the processing core. Furthermore, the number of instructions in a program is typically smaller in comparison to traditional load/store architectures where all operands have to be transferred to registers before any operations can be performed. There are 47 instructions of the following types:

- Load/store instructions for moving data between registers and memory, and loading of immediate values.
- Integer and fixed-point arithmetic operations.
- Bitwise logical and shift operations.
- Control flow instructions and comparison operations.
- Bus access instructions for read & write operations on an AMBA AHB bus (not covered in this paper).

The REDFIN processing core fetches instruction and data words from a small and fast on-chip SRAM. This only allows for execution of simple programs, however, it also eliminates the need to implement caches and thus removes a source of non-determinism of conventional processors. High performance is not one of the main goals, hence the core is not pipelined and does not need to resolve data/control hazards or perform any form of speculative execution. These properties greatly simplify worst-case execution time analysis.

2.2 Requirements for Formal Verification

Verification of *functional correctness* of REDFIN programs, as defined by a requirement specification, clearly is an essential task for the development of space electronics. There are also important *non-functional requirements*, such as worst-case execution time and energy consumption, which rely on the implementation guarantees provided by the processing core.

To reduce verification complexity, the REDFIN core only allows to execute a single subroutine whose execution is triggered by a higher-level controller in the system. The implementation guarantees that concurrent bus accesses to the processor registers or memory do not affect the subroutine execution time. Furthermore, the processor does not implement interrupt handling. All these measures are taken to provide real-time subroutine execution guarantees and make the verification of non-functional properties feasible.

Despite these restrictions the REDFIN core has already proven its effectiveness for simple control tasks and arithmetic computations as part of an antenna pointing unit for satellites. Nevertheless, verification can be difficult and time-consuming, even for small and simple programs. Verification activities, following engineering standards for space electronics, typically outweigh programming and design tasks by a factor of two in terms of development hours. Usually verification is performed via program execution on an instruction set simulator or a hardware model of the processor. Manually deriving test cases from the specification is cumbersome and error-prone and simulation times can become prohibitively long with a large number of tests that are often needed to reach the desired functional and code coverage. Formal verification methods can prove that a program satisfies certain properties for all possible test cases and are therefore immensely valuable for completing the verification with superior efficiency and quality.

3 MODELLING REDFIN IN HASKELL

In this section we formally define the REDFIN microarchitecture and express the semantics of the instruction set as an explicit and symbolic state transformer.

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

type Register           = SymbolicValue Word2
type Value              = SymbolicValue Int64
type RegisterBank       = SymbolicArray Word2 Int64
type MemoryAddress      = SymbolicValue Word8
type Memory             = SymbolicArray Word8 Int64

type InstructionAddress = SymbolicValue Word8
type InstructionCode    = SymbolicValue Word16
type Program            = SymbolicArray Word8 Word16

data Flag               = Condition | Overflow | Halt ...
type Flags              = SymbolicArray Flag Bool
type Clock              = SymbolicValue Word64
```

Fig. 2. Basic types for modelling REDFIN.

3.1 The REDFIN Microarchitecture State

The main idea of our approach is to use an explicit state transformer semantics of the REDFIN microarchitecture. The `State` of the entire processing core is a product of states of every component, see Fig. 2. We define `SymbolicValue` and `SymbolicArray` on top of the SBV library [9] that we use as a frontend for SMT translation and verification.

There are 4 registers (addressed by `Word2`) and 256 memory cells (addressed by `Word8`) that store 64-bit values (`Int64`). The register bank and memory are represented by symbolic arrays that can be accessed via SBV's functions `readArray` and `writeArray`. REDFIN uses 16-bit `InstructionCode`s, whose 6 leading bits contain the opcode, and the remaining 10 bits hold instruction arguments. The `Program` maps 8-bit instruction addresses to instruction codes.

The microarchitecture status `Flags` support conditional branching, track integer overflow, and terminate the program (we omit a few other flags for brevity). The `Clock` is a 64-bit counter incremented on each clock cycle. Status flags and the clock are used for diagnostics, formal verification, and worst-case execution time analysis.

3.2 Instruction and Program Semantics

We can now define the formal semantics of REDFIN instructions and programs as a *state transformer* $T : S \rightarrow S$, i.e. a function that maps states to states. We distinguish instructions and programs by using Haskell's list notation, e.g. $T_{\texttt{nop}}$ is the semantics of the instruction `nop` $\in I$, whereas $T_{\texttt{[nop]}}$ is the semantics of the single-instruction program `[nop]` $\in P$. [2]

[2]REDFIN does not have a dedicated `nop` instruction, but it can be expressed as a jump to the next instruction, i.e. `jmpi 0`.

Definition (program semantics): The semantics of a program $p \in P$ is inductively defined as follows:

The semantics of the *empty program* `[]` $\in P$ coincides with the semantics of the instruction `nop` and is the identity state transformer: $T_{\texttt{[]}} = T_{\texttt{nop}} = \texttt{id}$.

The semantics of a *single-instruction program* `[i]` $\in P$ is a composition of (i) fetching the instruction from the program memory T_{fetch}, (ii) incrementing the instruction counter T_{inc}, and (iii) the state transformer of the instruction itself $T_{\texttt{i}}$; or, using the order of state components from Fig. 2:

$$\begin{aligned} T_{fetch} &= (r, m, ic, ir, p, f, c) \mapsto (r, m, ic, p[ic], p, f, c + 1) \\ T_{inc} &= (r, m, ic, ir, p, f, c) \mapsto (r, m, ic + 1, ir, p, f, c) \\ T_{\texttt{[i]}} &= T_{\texttt{i}} \circ T_{inc} \circ T_{fetch} \end{aligned}$$

The semantics of a *composite program* `i:p` $\in P$, where the operator `:` prepends an instruction `i` $\in I$ to a program `p` $\in P$, is defined as $T_{\texttt{i:p}} = T_{\texttt{p}} \circ T_{\texttt{[i]}}$.

We represent state transformers in Haskell using the *state monad*, a classic approach to emulating mutable state in a purely functional programming language [20]. We call our state monad `Redfin` and define it as follows[3]:

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

A computation of type `Redfin a` yields a value of type `a` and possibly alters the `State` of the REDFIN microarchitecture. The type `Redfin ()` describes a computation that does not produce any value as part of the state transformation; such computations directly correspond to state transformers. For example, here is the state transformer T_{inc}:

```
incrementInstructionCounter :: Redfin ()
incrementInstructionCounter = Redfin $ \current -> ((), next)
  where
    next = current { instructionCounter = instructionCounter current + 1 }
```

In words, the state transformer looks up the value of the `instructionCounter` in the `current` state and replaces it in the `next` state with the incremented value. We can compose such primitive computations into more complex state transformers using Haskell's `do`-notation:

```
readInstructionRegister :: Redfin InstructionCode
readInstructionRegister = Redfin $ \s -> (instructionRegister s, s)

executeInstruction :: Redfin ()
executeInstruction = do
    fetchInstruction
    incrementInstructionCounter
    instructionCode <- readInstructionRegister
    decodeAndExecute instructionCode
```

Here `readInstructionRegister` reads the instruction code from the current state *without modifying it*, and is subsequently used in `executeInstruction`, which defines the semantics of the REDFIN execution cycle. We omit definitions of `fetchInstruction` and `decodeAndExecute` for brevity. The latter is a case analysis of 47 opcodes that returns the matching instruction. We discuss several instructions below.

[3]A generic version of this monad is available in the standard module `Control.Monad.State`.

3.2.1 Halting the Processor. The instruction halt sets the flag **Halt**, which stops the execution of the current subroutine until a new one is started by a higher-level system controller that resets **Halt**.

```
halt :: Redfin ()
halt = writeFlag Halt true
```

The auxiliary function writeFlag modifies the flag:

```
writeFlag :: Flag -> SymbolicValue Bool -> Redfin ()
writeFlag flag value = Redfin $ \s -> ((), s')
  where
    s' = s { flags = writeArray (flags s) (flagId flag) value }
```

In the rest of the paper we will use auxiliary functions readRegister, writeRegister, readState, etc.; they are simple state transformers defined similarly to writeFlag.

3.2.2 Arithmetics. The instruction abs is more involved: it reads a register and writes back the absolute value of its contents. The semantics accounts for the potential integer overflow that leads to the *negative resulting value* when the input is -2^{63} (REDFIN uses the common two's complement signed number representation). The overflow is flagged by setting **Overflow**. We use SBV's symbolic *if-then-else* operation ite to *merge* two symbolic values — in this case two possible next states, one of which is a state with the **Overflow** flag set:

```
abs :: Register -> Redfin ()
abs reg = do
    state  <- readState
    result <- fmap Prelude.abs (readRegister reg)
    let (_, state') = transform (writeFlag Overflow true) state
    writeState $ ite (result .< 0) state' state
    writeRegister reg result
```

3.2.3 Conditional Branching. As an example of a control flow instruction consider jmpi_ct, which tests the **Condition** flag, and adds the provided offset to the instruction counter if the flag is set.

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

We use our Haskell encoding of the state transformer as a *metalanguage*: we operate the REDFIN core as a puppet master, using external meta-notions of addition, comparison and let-binding. From the processor's point of view, we have infinite memory and act instantly, which gives us unlimited modelling power. For example, we can simulate the processor environment in an external tool and feed its result to writeRegister as if it was obtained in one clock cycle.

3.3 Symbolic simulation

Having defined the semantics of REDFIN programs, we can perform *symbolic processor simulation.*

There are many flavours of symbolic execution [2] and there is no single answer to the question of which one is the best. The choice of the symbolic execution technique depends heavily on the verification scenarios. The verification framework for REDFIN is designed to deal with two main classes of programs: (i) arithmetic calculations that are statically provable to be terminating and (ii) control programs with unbounded loops whose termination depend on dynamic input data. We approach verification of the former using *symbolic execution with merging* that allows for whole-program verification by translating the program and the verification condition into a single SMT formula. However, unconditional merging does not work for non-terminating programs. To verify those, we use an alternative approach to symbolic execution, which constructs tree-shaped traces with multiple execution paths that can be analysed and verified separately. The presented framework thus works in two modes — *merging* and *branching* — see examples in §4 and §5, respectively.

3.3.1 Merging Mode. The function `simulate` takes a number of simulation steps N and an initial symbolic `State` as input, and runs `executeInstruction` defined above N times. In each `state` we merge two possible futures: (i) if the `Halt` flag is set, we stop the simulation and remain in the current `state`, since in this case the processor must remain idle; (ii) otherwise we continue the simulation from the `future` state.

```
simulate :: Int -> State -> State
simulate steps state =
    let halted = readArray (flags state) (flagId Halt)
    in if steps <= 0
       then state
       else let future = snd (transform executeInstruction state)
            in  ite halted state (simulate (steps-1) future)
```

The function `ite` performs symbolic merging of two possible next states, depending on whether the program has `halted`. As we have seen, the semantics of individual instructions, as captured by `executeInstruction`, also uses `ite` to merge possible next states when encountering choices, thereby resulting in a linear simulation path.

3.3.2 Branching Mode. In the branching mode, on the contrary, we do not perform any merging at all and generate a tree of simulation paths where every node contains a program state with an associated *path condition* — a symbolic Boolean expression that encodes the choices taken along the path leading to the node.

To implement the branching semantics, we modify the `Redfin` monad by adding support for non-determinism and tracking of path conditions. While we omit the details, below we show how the types of the main simulation functions, `executeInstruction` and `simulate`, need to change. The former becomes a non-deterministic function that can return multiple next states:

```
executeInstruction :: (State, SymbolicValue Bool) -> [(State, SymbolicValue Bool)]
```

For example, it will return two states in the case of the conditional jump instruction `jmpi_ct` instead of merging them with `ite`. The path condition (of type `SymbolicValue Bool`) will be conjoined with the jump condition for the first returned state, and with its negation for the second.

The type of `simulate` becomes more involved too. Since the semantics of individual instructions now produces multiple next states, the symbolic execution trace naturally grows into a tree:

```
data Tree a = Node a [Tree a]
```

```
simulate :: Int -> (State, SymbolicValue Bool) -> Tree (State, SymbolicValue Bool)
```

The simulation starts form an initial state with a path condition representing the precondition of the program, which can be instantiated with `true` if no precondition is imposed. Proceeding further, more successor states can stem from every state, until either the `halt` instruction is encountered or the maximum number of simulation steps is reached.

Symbolic simulation is very powerful. It allows us to formally verify properties of REDFIN programs by fixing some parts of the state to constant values (e.g., the program), and then making assertions on the symbolic part of the resulting state, as demonstrated in sections §4 and §5.

4 FORMAL VERIFICATION

This section presents a formal verification framework developed on top of the REDFIN semantic core (§3). The verification workflow comprises the following steps:

- Develop programs in low-level REDFIN assembly, and in a high-level typed language embedded in Haskell.
- Test REDFIN programs on concrete input values.
- Define functional correctness and worst case execution time properties in the SBV language.
- Verify the properties or obtain counterexamples.

Consider the following simple spacecraft control task.

Let t_1 and t_2 be two different time points (measured in ms), and p_1 and p_2 be two power values (measured in mW). Calculate the estimate of the total energy consumption during this period using linear approximation, rounding down to the nearest integer:

$$energyEstimate(t_1, t_2, p_1, p_2) = \left\lfloor \frac{|t_1 - t_2| * (p_1 + p_2)}{2} \right\rfloor.$$

This task looks too simple, but in fact it has a few pitfalls that, if left unattended, may lead to the failure of the space mission. Examples of subtle bugs in seemingly simple programs leading to a catastrophe include 64-bit to 16-bit number conversion overflow causing the destruction of Ariane 5 rocket [3] and the loss of NASA's Mars orbiter due to incorrect unit conversion [16]. Let us develop and verify a REDFIN program for this task.

We can write programs in the untyped REDFIN assembly, or in a typed higher-level expression language. The former allows engineers to hand-craft highly optimised programs under tight resource constraints, while the latter brings type-safety and faster prototyping. We start with the high-level approach and define an expression that can be used both as a Haskell function and a high-level REDFIN expression:

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

Thanks to polymorphism, we can treat `energyEstimate` both as a numeric function, and as an abstract syntax tree that can be *compiled* into a REDFIN assembly `Script`. Due to the lack of space we omit the implementation of `Script`, but one can think of it as a restricted version of the `Redfin` state transformer, which we use to write *programs that can manipulate the processor state only by executing instructions*, e.g. the only way to set the `Overflow` flag is to execute an arithmetic instruction that might cause an overflow.

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

Here the type **IntegerVariable** is used to statically distinguish between integer and fixed-point numbers, **Temporary** to mark temporary words, so they cannot be mixed with inputs and outputs, and **Stack** to denote the location of the stack pointer. The **let** block declares six adjacent memory addresses: four input values $\{t_1, t_2, p_1, p_2\}$, a temporary word, and a stack pointer. We `compile` the high-level expression `energyEstimate` into the assembly language by translating it to a sequence of REDFIN instructions. The first argument of the `compile` function holds the register `r0` which contains the estimated energy value after the program execution.

We can run symbolic simulation for 100 steps, initialising the program and data memory of the processor using the function `simulate` defined above and a helper function `boot`.

```
main = do
    let dataMemory = [10, 5, 3, 5, 0, 100]
        finalState = simulate 100 $ boot energyEstimateHighLevel dataMemory
    printMemoryDump 0 5 (memory finalState)
    putStrLn $ "R0: " ++ show (readArray (registers finalState) r0)
```

As the simulation result we get a `finalState`. We inspect it by printing relevant components: the values of the first six memory cells, and the result of the computation located in the register `r0`. Note that the stack pointer (cell 5) holds 100, as in the initial state, which means the stack is empty.

```
Memory dump: [10, 5, 3, 5, 5, 100]
R0: 20
```

Simulating programs with specific inputs is useful for diagnostics and test, but SMT solvers allow us to verify the correctness for *all valid input combinations*. To demonstrate this, let us discover a problem in our energy estimation program. Consider the following correctness property.

Assuming that values p_1 and p_2 are non-negative integers, the energy estimation subroutine must always return a non-negative integer value.

To check that the program meets this requirement, we translate `energyEstimateHighLevel` into an SMT formula, and formulate the corresponding theorem:

```
theorem = do
    t1 <- forall "t1" -- Initialise symbolic variables
    t2 <- forall "t2"
    p1 <- forall "p1"
    p2 <- forall "p2" -- And then add constraints:
    constrain $ p1 .>= 0 &&& p2 .>= 0
    -- Initialise the data memory with symbolic variables:
    let dataMemory = [t1, t2, p1, p2, 0, 100]
        finalState = simulate 100 $ boot energyEstimateHighLevel dataMemory
        result = readArray (registers finalState) r0
        halted = readArray (flags finalState) (flagId Halt)
    return $ halted &&& result .>= 0 &&& result .== energyEstimate t1 t2 p1 p2
```

We extract the computed `result` and the value of the flag **Halt** from the `finalState`, and then assert that the processor has `halted`, and that the `result` is non-negative and is equal to that computed by the high-level Haskell expression `energyEstimate`. The resulting SMT formula can be checked by Z3 in 3.0s[4]:

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

Z3 has found a counterexample demonstrating that the program does not satisfy the above property. Indeed, the expression evaluates to a negative value on the provided inputs due to an *integer overflow*. We therefore refine the property:

According to the spacecraft power system specification, p_1 and p_2 are non-negative integers not exceeding 1W. The time is measured from the mission start, hence t_1 and t_2 are non-negative and do not exceed the time span of the mission, which is 30 years. Under these assumptions, the energy estimation subroutine must return a non-negative integer value.

We need to modify time and power constraints accordingly:

```
constrain $ p1 .<= toMilliWatts   ( 1 :: Watt)
        &&& t1 .<= toMilliSeconds (30 :: Year)
        &&& t1 .>= 0 &&& t2 .>= 0 &&& ... -- etc.
```

Rerunning Z3 produces the desired QED outcome in 4.8s.

The refinement has rendered the integer overflow impossible; in particular, `abs` can never be called with -2^{63} within the mission parameters. Such guarantee fundamentally requires solving an SMT problem, even if it is done at the type level, e.g. using *refinement types* [19].

The statically typed high-level expression language is very convenient for writing REDFIN programs, however, an experienced engineer can often find a way to improve the resulting code. In some resource-constrained situations, a fully hand-crafted assembly code may be required. As an example, consider the following low-level program:

```
energyEstimateLowLevel :: Script
energyEstimateLowLevel = do
    let { t1 = 0; t2 = 1; p1 = 2; p2 = 3 }
    ld r0 t1
    sub r0 t2
    abs r0
    ld r1 p1
    add r1 p2
    st r1 p2
    mul r0 p2
    sra_i r0 1
    halt
```

[4]We use a laptop with 2.90GHz Intel Core i5-4300U processor, 8GB RAM (3MB cache), and the SMT solver Z3 v4.5.1 (64-bit).

This program computes the energy estimate using only 9 instructions, whereas a direct unoptimised translation of the energyEstimate expression into assembly uses 79 instructions, most of them for stack manipulation.

To support the development of hand-crafted code, we use Z3 to *check the equivalence of REDFIN programs* by verifying that they produce the same output on all valid inputs. This allows an engineer to optimise a high-level prototype and have a guarantee that no bugs were introduced in the process.

```
equivalence = do
    t1 <- forall "t1"
    t2 <- forall "t2"
    p1 <- forall "p1"
    p2 <- forall "p2"
    constrain $ p1 .>= 0 &&& p2 .>= 0
            &&& t1 .>= 0 &&& t2 .>= 0
            &&& p1 .<= toMilliWatts   ( 1 :: Watt)
            &&& p2 .<= toMilliWatts   ( 1 :: Watt)
            &&& t1 .<= toMilliSeconds (30 :: Year)
            &&& t2 .<= toMilliSeconds (30 :: Year)
    let memory   = [t1, t2, p1, p2, 0, 100]
        llState  = simulate 100 $ boot energyEstimateLowLevel  memory
        hlState  = simulate 100 $ boot energyEstimateHighLevel memory
        llResult = readArray (registers llState) r0
        hlResult = readArray (registers hlState) r0
    return $ llResult .== hlResult
```

The equivalence check succeeds and takes 11.5s.

Every call of the executeInstruction function advances the clock field of the State (see Fig. 2) by the appropriate number of cycles, precisely matching the hardware implementation. This allows us to perform *best/worst-case execution timing analysis* using the optimisation facilities of SBV and Z3. As an example, let us determine the minimum and maximum number of clock cycles required for executing energyEstimateLowLevel. To make this example more interesting, we modified the semantics of the instruction abs and added 1 extra clock cycle in case of a negative argument.

```
timingAnalysis = optimize Independent $ do
  ... -- Initialise and run symbolic simulation
  minimize "Best case"  (clock finalState)
  maximize "Worst case" (clock finalState)
```

The total delay of the program depends only on the sign of $t_1 - t_2$, thus the best and worst cases differ only by one clock cycle. The worst case is achieved when the difference is negative ($t_1 - t_2 = -2$), as shown below. Z3 finishes in 0.5s.

```
Objective "Best case":
Optimal model:
  t1        = 549755813888
  t2        =  17179869184
  p1        =            0
  p2        =            0
  Best case =           12
```

```
Objective "Worst case":
Optimal model:
  t1         = 65535
  t2         = 65537
  p1         =     0
  p2         =     0
  Worst case =    13
```

5 SYMBOLIC EXECUTION IN PRESENCE OF UNBOUNDED LOOPS

Many programs targeting REDFIN share the distinctive feature of the energy estimation program considered in section §4, i.e. the existence of an *upper bound on execution time*, since their termination does not depend on input data. However, other programs may have a loop which is guarded by a termination condition that involves computation considering the input parameters of the program, thus making the loop *unbounded*.

Presence of unbounded loops makes program verification by symbolic execution considerably harder [2, p. 50:20], since the number of program execution paths becomes infinite. In this section we consider an example of a control program that drives a stepper motor and verify one of its essential safety properties by formulating it as a *loop invariant* and ensuring that the invariant holds for every possible state of the loop.

5.1 Stepper Motor Control Program

Stepper motors are often deployed as parts of antenna and solar panel pointing units in space satellites. We consider a program for controlling a motor with one degree of freedom. The control Algorithm 1 takes three input parameters:

- $dist$ — the distance to move the motor
- v_{max} — the maximal permitted velocity
- a_{max} — the maximal permitted acceleration

and computes a series of displacement and velocity values that will be used to move the motor. Since the algorithm is designed for controlling a stepper motor, the calculations happen in *discrete time*, i.e. every iteration of the *while* loop corresponds to a time interval; thus the deceleration (i.e. braking) distance is computed as

$$s_{decel} = a_{max} \cdot \frac{decel\_steps \cdot (decel\_steps + 1)}{2},$$

where $decel\_steps = \frac{v}{a_{max}}$ is the number of decelerating iterations needed for a full stop.

The conditional statement in line 9 decides whether to accelerate, to keep the velocity, or to decelerate; see Fig. 4 for example plots of velocity and distance travelled against time. The spike at the bottom-right of the velocity plot illustrates the edge case covered by the conditional statement on line 18: if the velocity is zero, but the target distance has not yet been reached, the motor must be moved further.

To deploy the Algorithm 1 to REDFIN, it has been manually implemented in REDFIN assembly. The resulting assembly program comprises 85 lines of code and closely mirrors the high-level pseudocode. Fig. 5 shows a fragment of the program's symbolic execution tree that corresponds to the decision whether to accelerate, keep the velocity, or decelerate the motor.

The decision is performed by computing the resulting total distance travelled from start to stop, based on the action taken in the current time step. First, the total distance is computed if the motor were to accelerate for one more time step and then decelerate in the subsequent time steps. If the computed total distance is less than or equal to *dist*, the decision to accelerate is committed. Otherwise, the algorithm checks whether the targeted distance can be met by maintaining the current velocity for one more time step. If even that would cause an overshoot, the decision for immediately commencing deceleration is taken. Fig. 4 illustrates this decision process by plotting the velocity and distance over time for a specific simulation run.

Algorithm 1 Motor Control Algorithm.

```
Input: $dist, v_{max}, a_{max}$
1: $s \leftarrow 0$
2: $v \leftarrow 0$
3: while $true$ do
4:     $decel\_steps \leftarrow v/a_{max}$                                    ▷ Compute deceleration distance
5:     $s_{decel} \leftarrow a_{max} \cdot decel\_steps \cdot (decel\_steps + 1)/2$   ▷ based on the current velocity
6:     if $decel\_steps \cdot a_{max} \neq v$ then
7:         $s_{decel} \leftarrow s_{decel} + v$
8:     $v_{next} = min(v_{max}, dist, v + a_{max})$
9:     if $s + s_{decel} + v_{next} \leq dist$ then
10:        $v \leftarrow v_{next}$                                            ▷ Accelerate
11:    else if $s + s_{decel} + v \leq dist$ then
12:        $v \leftarrow v$                                                   ▷ Keep velocity
13:    else                                                                   ▷ Decelerate
14:        if $v > decel\_steps \cdot a_{max}$ then
15:            $v \leftarrow decel\_steps \cdot a_{max}$
16:        else
17:            $v \leftarrow v - a_{max}$
18:    if $v = 0$ then
19:        if $s \neq dist$ then                                             ▷ Accelerate again to reach target
20:            $v \leftarrow min(dist - s, a_{max})$
21:        else
22:            $break$                                                        ▷ Terminate execution
23:    $s \leftarrow s + v$
```

5.2 Loop Invariant Verification

In order to ensure that the motor will not introduce disturbances and will not lead the whole unit out of its normal mode of operation, the velocity and acceleration of the motor must be kept within safe limits. This verification condition is motivated by the correctness requirements of the whole space satellite unit.

More formally, the verification condition means that at any iteration t of the loop the values of the expressions v^t, velocity, and $\left|v^t_{next} - v^t\right|$, acceleration, must never exceed the parameters v_{max} and a_{max}, respectively. This property is the loop invariant for the motor control program which ensures that velocity and acceleration always stay within their safe bounds. We formalise it as the following predicate that universally quantifies over the program's inputs and the loop's state:

$$\forall\ v_{max}\ a_{max}\ t\ v^t\ v^t_{next},\ v^t \leq v_{max} \wedge \left|v^t_{next} - v^t\right| \leq a_{max}$$

Fig. 3. Motor control loop invariant.

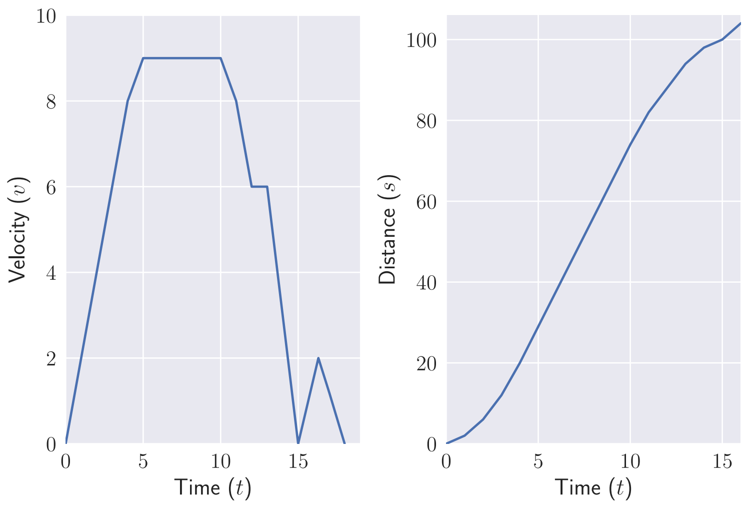

Fig. 4. Velocity (v) and distance travelled (s) plotted against time (t)

We will verify the loop invariant by using the verification framework in the *branching* mode §3. While symbolic execution with merging, which is implemented by the framework's *merging* mode, allows for intuitive formulation of properties for whole-program verification and is very useful for verifying finite programs, as we have reported in the section §4, in the presence of branches guarded by symbolic values, it suffers from *symbolic non-termination*; thus, for verifying the loop invariant, we rely on the branching mode of symbolic execution.
We take the following generic approach:

- Obtain the binary tree-shaped trace by symbolic execution in *branching* mode.
- Split the trace into linear paths, thus enumerating all possible execution scenarios.
- For every path perform the analysis:
  - extract the relevant parts of the state from the *last* node in the path, i.e. symbolic expressions stored representing s, v and v_{next};
  - extract the terminal *path condition* ϕ from the *last* node in the path;
  - construct a symbolic expression representing the *verification condition* ψ (Fig. 3);
  - verify the property in the given path by checking the following formula for satisfiability: $\phi \land \neg\psi$, i.e. the path condition conjoined with negated verification condition.
- The property holds if and only if for every path the solver returns Unsat, i.e. there are no assignments of the variables which satisfy the *negation* of the property to check, considering the terminal path condition.

As illustrated by Fig. 5, every conditional jump instruction produces two branches in the symbolic execution tree: the one where the current path condition is conjoined with the jump's guard and the one where it is conjoined with the guard's negation. However, if the resulting conjunction is unsatisfiable, the corresponding branch need not to be explored and can be safely pruned. Thus the symbolic execution engine needs to call an SMT solver every time a conditional jump is encountered to check if the path conditions of the branches are satisfiable.

Checking satisfiability of path conditions is essential for mitigating the path explosion problem. Precondition, when available, is assigned as the initial path condition and thus will become a

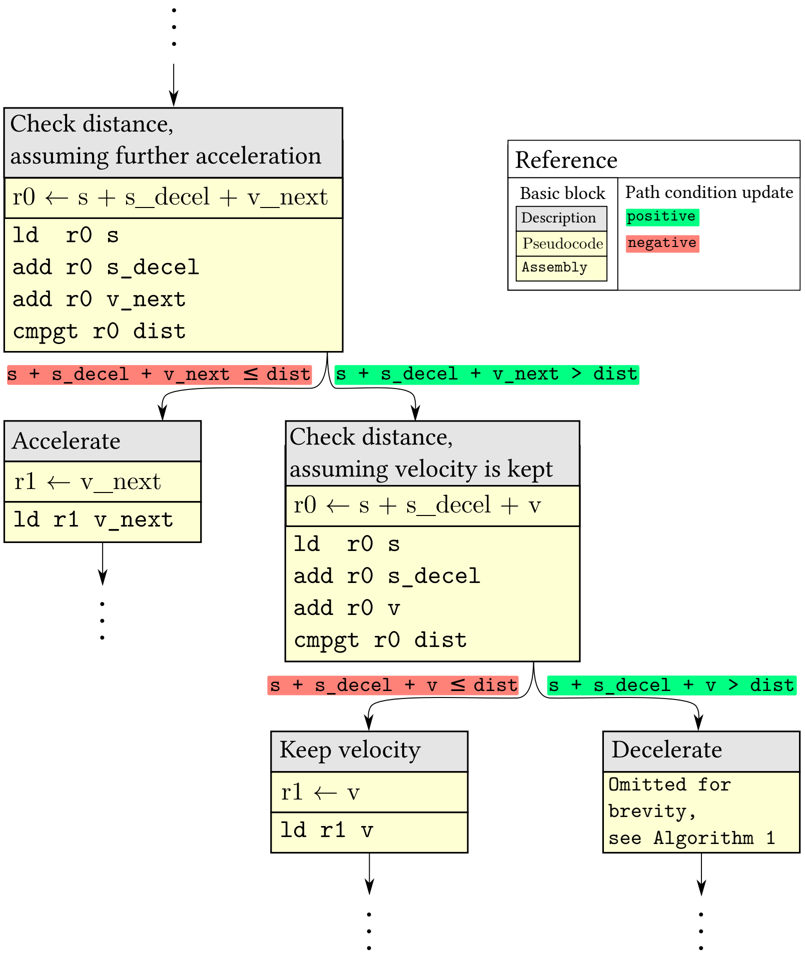

Fig. 5. Symbolic execution tree of a code fragment with conditional branching.

subterm of every formula submitted to the solver. By identifying strong preconditions, we can drastically reduce the number of satisfiable paths in the symbolic execution tree of a program that is being verified, thereby significantly shortening verification times.

With the branch pruning optimisation in place and the parameters restricted to $dist \in [1, 200]$, $v_{max} \in [1, 30]$ and $a_{max} \in [1, 30]$, the verification of the loop invariant takes around 40 minutes.

6 DISCUSSION

As the previous section §4 demonstrates, the presented approach provides a unified specification, testing, and formal verification framework. It allows the REDFIN engineering team to co-develop REDFIN software and hardware, by extending and modifying the default instruction semantics. By using Haskell as a metalanguage, one can implement higher-level languages on top of the REDFIN assembly, such as our simple statically-typed language for arithmetic expressions.

Static typing, polymorphism, do-notation, and availability of a mature symbolic manipulation library (SBV) were the key factors for choosing Haskell for this project. We also have a prototype implementation in the dependently-typed language Idris [4] that allows us to verify more sophisticated properties at the type level, however at the time of writing there is no equivalent of the SBV library in Idris, which is a significant practical disadvantage.

The `Script` monad was engineered to provide familiar assembly mnemonics and directives (e.g. labels), which allows engineers to start using the framework for developing REDFIN programs even without prior Haskell experience, hopefully increasing the uptake of the framework.

Thanks to symbolic simulation, we can uniformly handle both concrete and symbolic values, reusing the same code base and infrastructure for testing and formal verification. Testing yields trivial SMT problems that can be solved in sub-second time. Formal verification is more expensive: in our experiments, realistic programs (e.g. for controlling a stepper motor in antenna and solar panel positioning units, with a loop and 85 instructions) required 40 minutes, but one can easily construct tiny programs that will grind any SMT solver to a halt: for example, analysis of a single multiplication instruction can take half an hour if it is required to factor 64-bit numbers — try to factor 4611686585363088391 with an SMT solver! In such cases, conservatively proving some of the correctness properties at the type level can significantly increase the productivity. As a microbenchmark, we verified the correctness of an array summation program, reporting the number of SMT clauses and Z3 runtime for low-level (LL) and high-level (HL) programs:

Benchmark	Array size	Clauses LL	Clauses HL	Time LL	Time HL
Overflow: values not constrained	9	286	260	1.482s	0.443s
	12	492	453	3.604s	1.365s
	15	740	688	49.969s	7.362s
	18	1030	965	76.757s	88.458s
No overflow: all values in $[1, 1000]$	9	318	292	0.467s	0.119s
	12	549	510	0.739s	0.682s
	15	828	776	1.839s	6.408s
	18	1155	1090	9.944s	72.880s
Equivalence to the `sum` function	9	258	261	0.039s	0.097s
	12	495	459	0.053s	0.647s
	15	708	702	0.311s	7.322s
	18	1005	990	2.633s	71.896s

Table 1. Verification performance for an array summation program

The example of a stepper motor control program considered in section §5 proved challenging to verify. We have been able to ensure that a loop invariant representing an important safety property of the program holds, but had to restrict the input parameter space; we are working towards addressing the performance restrictions of the framework in order to be able to verify functional correctness of the motor control program, which involves exploring much larger path space than the loop invariant verification condition.

7 RELATED WORK

There is a vast body of literature available on the topic of formal verification, including verification of hardware processing cores and low-level software programs. Our work builds in a substantial way on a few known ideas that we will review in this section. We thank the formal verification and programming languages communities and hope that the formal semantics of the REDFIN processing core will provide a new interesting benchmark for future studies.

We model the REDFIN microarchitecture using a *monadic state transformer metalanguage* – an idea with a long history. Fox and Myreen [10] formalise the Arm v7 instruction set architecture in HOL4 and give a careful account to bit-accurate proofs of the instruction decoder correctness. Later, Kennedy et al. [12] formalised a subset of the x86 architecture in Coq, using monads for instruction execution semantics, and do-notation for assembly language embedding. Degenbaev [7] formally specified the *complete* x86 instruction set – a truly monumental effort! – using a custom domain-specific language that can be translated to a formal proof system. Arm's Architecture Specification Language (ASL) has been developed for the same purpose to formalise the Arm v8 instruction set [18]. The SAIL language [1] has been designed as a generic language for ISA specification and was used to specify the semantics of ARMv8-A, RISC-V, and CHERI-MIPS. Our specification approach is similar to these three works, but we operate on a much smaller scale of the REDFIN core and focus on verifying whole programs.

Our metalanguage is embedded in Haskell and does not have a rigorous formalisation, i.e. we cannot prove the correctness of the REDFIN semantics itself, which is a common concern, e.g. see Reid [17]. Moreover, our verification workflow mainly relies on *automated* theorem proving, rather than on *interactive* one. This is motivated by the cost of precise proof assistant formalisations in terms of human resources: automated techniques are more CPU-intensive, but cause less "human-scaling issues" [18]. Our goal was to create a framework that could be seamlessly integrated into an existing spacecraft engineering workflow, therefore it needed to have as much proof automation as possible. The automation is achieved by means of *symbolic program execution*. Currie et al. [5] applied symbolic execution with uninterpreted functions to prove equivalence of low-level assembly programs. The framework we present allows not only proving the equivalence of low-level programs, but also their compliance with higher-level specifications written in a subset of Haskell.

Finally, we would like to acknowledge the projects and talks that provided an initial inspiration for this work: the 'Monads to Machine Code' compiler by Diehl [8], RISC-V semantics by MIT [14], the assembly monad by Wall [21], and SMT-based program analysis by Jelvis [11].

ACKNOWLEDGEMENTS

We would like to thank Vitaly Bragilevsky, Georgi Lyubenov, Neil Mitchell, Charles Morisset, Artem Pelenitsyn, Danil Sokolov, as well as the three Haskell Symposium reviewers for their helpful feedback on an earlier version of this paper.

REFERENCES

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290384

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[3] Mordechai Ben-Ari. 2001. The Bug That Destroyed a Rocket. *SIGCSE Bull.* 33, 2 (June 2001), 58–59.

[4] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23 (9 2013), 552–593. Issue 05.

[5] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. 2006. Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions. *International Journal of Parallel Programming* 34, 1 (2006), 61–91.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[7] Ulan Degenbaev. 2012. *Formal specification of the x86 instruction set architecture.* Ph.D. Dissertation. Saarland University.

[8] Stephen Diehl. 2017. Monads to Machine Code. https://web.archive.org/web/20171207020256/http://www.stephendiehl.com/posts/monads_machine_code.html.

[9] Levent Erkok. 2019. *SBV: SMT Based Verification in Haskell.* http://leventerkok.github.io/sbv/

[10] Anthony Fox and Magnus O Myreen. 2010. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *International Conference on Interactive Theorem Proving*. Springer, 243–258.

[11] Tikhon Jelvis. 2016. Analyzing Programs with Z3 (video recording of Compose Conference talk). http://jelv.is/talks/compose-2016.

[12] Andrew Kennedy, Nick Benton, Jonas B Jensen, and Pierre-Evariste Dagand. 2013. Coq: the world's best macro assembler?. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. ACM, 13–24.

[13] Nancy G. Leveson. 2004. Role of Software in Spacecraft Accidents. *Journal of Spacecraft and Rockets* 41, 4 (2004), 564–575.

[14] MIT. 2017. A formal specification of the RISC-V ISA written in Haskell. https://github.com/mit-plv/riscv-semantics.

[15] Andrey Mokhov, Georgy Lukyanov, and Jakob Lechner. 2019. Formal Verification of Spacecraft Control Programs (Experience Report). In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. ACM, New York, NY, USA, 139–145. https://doi.org/10.1145/3331545.3342593

[16] NASA. 1999. *Mars Climate Orbiter Mishap Investigation Board Phase I Report*. Technical Report.

[17] Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the Arm V8-m Architecture Specification. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 88:1–88:24.

[18] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of processors with ISA-Formal. In *International Conference on Computer Aided Verification*. Springer, 42–58.

[19] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 269–282.

[20] Philip Wadler. 1990. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 61–78.

[21] Lewis Wall. 2017. An ASM Monad. http://wall.org/~lewis/2013/10/15/asm-monad.html.