# Calyx + HardKAT: A Verified IR for Calyx - CS 6861 Final Project

Jan-Paul Ramos-Dávila

-

## 1 INTRODUCTION

The following pipeline is a proof of concept for verifying the naive `par` operator in Calyx — an infrastructure for specifying hardware accelerators — using a Kleene Algebra framework (aided by the already existing HardKAT project). It's a common occurrence among developers that while attempting to implement more robust formal methods to verify their domain specific languages — specifically those of the form of symbolic execution algorithms — it tends to be difficult to reason about. The main issues we seek to tackle in this project are:

- Ameliorate the issue of depending on various approximations and heuristics in symbolic execution algorithms, which can lead to less precise results.
- Help developers understand non-carefully implemented algorithms, which might produce non-rigorous results.

The implementation can be found at this GitHub repository. Instructions for setup and prerequisites are specified in the README.

### 1.1 HardKAT

HardKAT [1] (Hardware Kleene Algebra with Tests) is a formal framework designed to model and verify hardware accelerator designs. It leverages the equational theory of Kleene Algebra with Tests (KAT) to describe and reason about both sequential and concurrent hardware designs. HardKAT's concurrency model extends the traditional KAT with parallel composition and global state manipulations, allowing it to handle more complex and larger designs.

The following BNF describes the syntax for the subset of HardKAT which deals with concurrency:

$$\text{Global State} \quad g ::= g_1 \mid \cdots \mid g_n \tag{1}$$

$$\text{Observations} \quad a, v ::= \top \mid \bot \mid g_n = n \mid a \wedge b \mid a \vee b \mid \neg a \tag{2}$$

$$\text{Global Actions} \quad g_1 \mapsto n \tag{3}$$

$$\text{Concurrency} \quad p \parallel q \tag{4}$$

*1.1.1 (Example) Bluespec SystemVerilog.* BSV is a high-level hardware description language that provides abstractions for designing and simulating hardware. We can use BSV to show how it can be described and verified in HardKAT.

```
module mkMux (input Bit in0, input Bit in1,
              input Bit sel, output Bit out);
  rule mux_rule;
    out <= sel ? in1 : in0;
  endrule
endmodule
```

**Listing 1: 1-bit Multiplexer**

*1.1.2 Global State and Observations.* Let $g$ represent the global state of the hardware.

Observations are characterized as follows:

- $g_{in0} = 0$ or $g_{in0} = 1$: State of input $in0$.
- $g_{in1} = 0$ or $g_{in1} = 1$: State of input $in1$.
- $g_{sel} = 0$ or $g_{sel} = 1$: State of select line $sel$.
- $g_{out} = 0$ or $g_{out} = 1$: State of output $out$.

Actions are characterized as follows:
Assign $g_{out}$ based on $g_{sel}$:

- $\mid g_{out} \mapsto g_{in0}$ if $g_{sel} = 0$
- $\mid g_{out} \mapsto g_{in1}$ if $g_{sel} = 1$

In this simple example, there are no concurrent actions, but HardKAT can handle parallel composition of actions:

- $\mid g_{out} \mapsto g_{in0} \parallel g_{sel} = 0$
- $\mid g_{out} \mapsto g_{in1} \parallel g_{sel} = 1$

### 1.2 Calyx

In Calyx [2], concurrency is managed through `par` blocks, which allow parallel execution of groups but do not guarantee any specific scheduling. This results in non-deterministic behavior, particularly when there are potential data races. Here are the key definitions related to concurrency in Calyx:

- *Group*: A basic unit of computation in Calyx, defining a set of actions to be executed together.
- *Parallel Block* (`par`) : A control structure that allows multiple groups to execute in parallel.
- *Data Race*: Occurs when two or more groups access the same data item, with at least one access being a write, leading to undefined behavior.

```
x = std_reg(32);
y = std_reg(32);

group g1 {
  x.in = y.out;
  x.write_en = 1'd1;
  g1[done] = x.done;
}
group g2 {
  y.in = x.out;
  y.write_en = 1'd1;
  g2[done] = y.done;
}

control {
  par {
    g1;
    g2;
  }
}
```

**Listing 2: Calyx program with non-determinism due to `par` blocks**

In this program, g1 and g2 can execute in parallel, but without any synchronization, leading to non-deterministic results.

## 2 AXIOMATIZATIONS

We take it to condense the overall HardKAT framework into a set of theorems which match up with the complexities of the Calyx par construct. Firstly, the code and theorems for HardKAT are still unpublished, and although the authors have graciously provided feedback during this process, we must extract specific theorems which do not necessarily hold in the overall HardKAT framework. Secondly, the general framework has severe implementation issues for building front ends with different HDL's (for example, BSV and Calyx have significantly different behaviors for non-determinism as will be described in 1.2). Therefore, we take these axiomatizations as non-trivially different from the overall HardKAT framework.

- Sequential Composition:

$$(p \cdot q) \cdot r = p \cdot (q \cdot r) \quad \text{(Associativity)}$$
$$p \cdot 1 = p \quad \text{(Identity)}$$

- Parallel Composition:

$$p \parallel q = q \parallel p \quad \text{(Commutativity)}$$
$$(p \parallel q) \parallel r = p \parallel (q \parallel r) \quad \text{(Associativity)}$$
$$p \parallel (q + r) = (p \parallel q) + (p \parallel r) \quad \text{(Distributivity over Union)}$$
$$p \parallel 0 = p \quad \text{(Identity)}$$

- Observations:

$$a \wedge \top = a$$
$$a \vee \bot = a$$
$$\neg(\neg a) = a$$
$$a \wedge \neg a = \bot$$

### 2.1 Relating HardKAT and Calyx's Concurrency Semantics

*2.1.1 (Annotation) Undefined Behavior for Data Races.* If $p \parallel q$ results in a data race, it leads to an undefined state $\epsilon$.

*2.1.2 (Annotation) Non-Deterministic Communication.* HardKAT can model non-deterministic communication (this is clarified in the next section) between par arms without guarantees of specific visibility.

*2.1.3 (Theorem) Correctness of Mux.* If $g_{sel} = 0$, then $g_{out} = g_{in0}$. Similarly, if $g_{sel} = 1$, then $g_{out} = g_{in1}$.

$$(g_{sel} = 0 \wedge g_{out} = g_{in0}) \vee (g_{sel} = 1 \wedge g_{out} = g_{in1})$$

is an invariant.

### 2.2 HardKAT in the context of Calyx's Concurrency

Using HardKAT's equational theory, we can model the concurrency semantics of Calyx's par blocks

*2.2.1 (Theorem) Correctness with Synchronization.* Consider the following Calyx program:

```
x = std_reg(32);
y = std_reg(32);

group g1 {
  x.in = y.out;
  x.write_en = 1'd1;
  g1[done] = x.done;
}
group g2 {
  y.in = x.out;
  y.write_en = 1'd1;
  g2[done] = y.done;
}

control {
  par {
    seq {
      g1;
      @sync(1) g1_sync;
    }
    seq {
      g2;
      @sync(1) g2_sync;
    }
  }
}
```

**Listing 3: Calyx program with synchronization**

Using HardKAT, we can express the synchronization as follows:

- $g_1 \parallel g_2$: Parallel execution of g1 and g2.
- $|g_x \mapsto g_y \parallel |g_y \mapsto g_x$: Potential data race without synchronization.
- Synchronization points ensure visibility:

$$g_x = g_y \quad \text{(Before g1\_sync and g2\_sync)}$$

*2.2.2 (Theorem) For the mux example.* If $g_{sel} = 0$, then $g_{out} = g_{in0}$. Similarly, if $g_{sel} = 1$, then $g_{out} = g_{in1}$.

$$(g_{sel} = 0 \wedge g_{out} = g_{in0}) \vee (g_{sel} = 1 \wedge g_{out} = g_{in1})$$

is an invariant.

By incorporating synchronization points using @sync, we ensure that concurrent groups in Calyx can safely communicate and observe events, thereby eliminating non-determinism and ensuring predictable behavior. Note this is not the current implementation of Calyx, as parallelism does not have a formal semantics yet. Thus, encompassing non-determinism is essential for implementation. However, section 4.2 discusses a more nuance implementation with the assumption of a formalized sync operator.

*2.2.3 Calyx's Symbolic Executor.* The symbolic execution framework translates Calyx programs into their Racket DSL equivalents, enabling verification of a broader set of hardware architectures. Currently, parallel execution is handled using the 'par-to-seq' pass, which sequentializes parallel blocks to simplify verification. For example, pipelines, a common form of hardware parallelism, can be verified by sequentializing their stages. Inline pass inlining simplifies the verification of subcomponents by flattening their hierarchical structure. To ensure end-to-end verification, the system validates each compiler pass to ensure it preserves functional correctness. This involves comparing the initial and final programs using symbolic execution and SMT solvers to prove equivalence.

## 3 IMPLEMENTATION

The following pipeline defines a robust Calyx-to-HardKAT translator, integrate it into the Calyx compiler, and use symbolic execution

to verify the correctness of hardware accelerator designs. This approach leverages the equational theory of HardKAT to handle both sequential and concurrent constructs.

## 3.1 Parser

This parser module reads Calyx source code and generate an AST representation of the program.

```
use calyx::parser::{parse_program};
use calyx::ast::{Program, Group};

fn parse_calyx_program(source: &str) -> Program {
parse_program(source).unwrap()
}
```

**Listing 4: Parser Snippet**

## 3.2 Translator

A Rust module translates the Calyx AST into HardKAT Intermediate Representation (IR). The translation rules for the sequential and concurrent constructs following the previous definitions. It converts the parsed Calyx AST into HardKAT IR, handling both sequential and concurrent constructs.

```
use calyx::ast::{Program, Group, Control};
use hardkat::ir::{HardKATProgram, HardKATGroup};

fn translate_to_hardkat(program: Program) ->
     HardKATProgram {
let mut hk_program = HardKATProgram::new();

csharp
Copy code
for group in program.groups {
    let hk_group = translate_group(group);
    hk_program.add_group(hk_group);
}

hk_program
}

fn translate_group(group: Group) -> HardKATGroup {
let mut hk_group = HardKATGroup::new(group.name);

scss
Copy code
for action in group.actions {
    let hk_action = translate_action(action);
    hk_group.add_action(hk_action);
}

hk_group
}

fn translate_action(action: Action) -> HardKATAction {
match action {
Action::Seq(actions) => {
let hk_actions = actions.into_iter().map(translate_action
     ).collect();
HardKATAction::Seq(hk_actions)
},
Action::Par(actions) => {
let hk_actions = actions.into_iter().map(translate_action
     ).collect();
HardKATAction::Par(hk_actions)
},
// Handle other action types...
}
```

**Listing 5: Translator Snippet**

## 3.3 Integration

We then extend the Calyx compiler to include a pass for translating to HardKAT IR. This ensures that the translated IR can be processed by the subsequent stages of the Calyx compiler.

```
use calyx::passes::{Pass};
use hardkat::ir::{HardKATProgram};

struct HardKATPass;

impl Pass for HardKATPass {
fn run(&self, program: &mut Program) -> Result<(), String
     > {
let hk_program = translate_to_hardkat(program.clone());
program.set_hardkat_ir(hk_program);
Ok(())
}
}
```

**Listing 6: Calyx Compiler Snippet**

## 3.4 Symbolic Execution

The symbolic execution tool executes Calyx programs symbolically, tracking the state of each variable and the conditions under which each state transition occurs.

```
use calyx::sexp::{SymbolicExecutor};

fn symbolic_execute(program: &Program) {
let executor = SymbolicExecutor::new(program);
executor.run();
}
```

**Listing 7: Symbolic Execution Snippet**

# 4 RUNNING EXAMPLE

Consider the following example of a Calyx program that demonstrates parallelism and the proposed `@sync` annotation for synchronization:

```
component main() -> () {
  cells {
    x = std_reg(32);
    y = std_reg(32);
  }
  wires {
    group g1 {
      x.in = y.out;
      x.write_en = 1'd1;
      g1[done] = x.done;
    }
    group g2 {
      y.in = x.out;
      y.write_en = 1'd1;
      g2[done] = y.done;
    }
  }
  control {
    par {
```

```
      seq {
        g1;
        @sync(1) g1;
      }
      seq {
        g2;
        @sync(1) g2;
      }
    }
  }
}
```

**Listing 8: Calyx Program with @sync**

This program attempts to swap the values of two registers, x and y, in parallel. The `@sync(1)` annotation ensures that the events from both parallel sequences are visible to each other.

## 4.1 Translation to HardKAT IR

The Calyx program can be translated into HardKAT IR using the defined translation rules. The translated HardKAT IR will reflect the parallel execution and synchronization semantics. The translated HardKAT IR ensures that the parallel execution semantics of the Calyx program are preserved and that synchronization points are correctly handled.

```
HardKATProgram {
  groups: [
    HardKATGroup {
      name: "g1",
      actions: [
        HardKATAction::Assign("x.in", "y.out"),
        HardKATAction::Assign("x.write_en", "1'd1"),
        HardKATAction::Check("x.done")
      ]
    },
    HardKATGroup {
      name: "g2",
      actions: [
        HardKATAction::Assign("y.in", "x.out"),
        HardKATAction::Assign("y.write_en", "1'd1"),
        HardKATAction::Check("y.done")
      ]
    }
  ],
  control: HardKATControl::Par(
    HardKATControl::Seq(
      vec![
        HardKATAction::Group("g1"),
        HardKATAction::Sync(1)
      ]
    ),
    HardKATControl::Seq(
      vec![
        HardKATAction::Group("g2"),
        HardKATAction::Sync(1)
      ]
    )
  )
}
```

**Listing 9: HardKAT IR**

## 4.2 More on Non-Determinism in Calyx

Calyx's current parallel semantics allow for non-deterministic execution of par blocks, which can lead to undefined behavior due to data races. The @sync annotation is a proposed solution to introduce synchronization points in parallel programs, ensuring that certain events are visible across parallel threads.

An alternative approach to addressing non-determinism could involve introducing explicit communication channels or memory fences that enforce order and visibility constraints between parallel threads. This approach would provide stronger guarantees about the execution order and state visibility in parallel programs.

*4.2.1 (Example) Here is a concrete example of how the* `@sync` *annotation can be used to ensure correct synchronization in a parallel Calyx program.*

```
component main() -> () {
  cells {
    x = std_reg(32);
    y = std_reg(32);
  }
  wires {
    group g1 {
      x.in = y.out;
      x.write_en = 1'd1;
      g1[done] = x.done;
    }
    group g2 {
      y.in = x.out;
      y.write_en = 1'd1;
      g2[done] = y.done;
    }
  }
  control {
    par {
      seq {
        g1;
        @sync(1) g1;
      }
      seq {
        g2;
        @sync(1) g2;
      }
    }
  }
}
```

**Listing 10: Calyx Program with Explicit Synchronization**

# 5 FORMALISMS

Now we want to handwritten proofs to verify the validity of our newly implemented semantics. The following are axiomatizations for HardKAT from standard KA semantics:

$$1 + x^*x \leq x^*$$
$$1 + xx^* \leq x^*$$
$$b + ax \leq x \implies a^*b \leq x$$
$$b + xa \leq x \implies ba^* \leq x$$

For parallel composition ($\|$):

(1) $p \parallel q = q \parallel p$ (Commutativity)
(2) $(p \parallel q) \parallel r = p \parallel (q \parallel r)$ (Associativity)
(3) $p \parallel (q + r) = (p \parallel q) + (p \parallel r)$ (Distributivity over +)
(4) $p \parallel 0 = p$ (Identity)
(5) If $p \parallel q$ results in a data race, it leads to an undefined state $\epsilon$.

## 5.1 Formalizing states

*5.1.1 Global State.*

$$g ::= g_1 \mid \cdots \mid g_n$$

*5.1.2 Observations.*

$$a, v ::= \top \mid \bot \mid g_n = n \mid a \wedge b \mid a \vee b \mid \neg a$$

*5.1.3 Global Actions.*

$$\mid g_1 \mapsto n$$

## 5.2 Proofs of Basic Axioms in KAT

*5.2.1* $1 + x^*x \leq x^*$.

Proof. By definition of the Kleene star ($x^*$):

$$1 + x^*x \leq x^*$$

□

*5.2.2* $1 + xx^* \leq x^*$.

Proof. By definition of the Kleene star ($x^*$):

$$1 + xx^* \leq x^*$$

□

*5.2.3* $b + ax \leq x \implies a^*b \leq x$.

Proof. Assume $b + ax \leq x$. We need to show $a^*b \leq x$. Base case ($n = 0$):

$$a^0 b = 1 \cdot b = b \leq x \quad \text{(by assumption)}$$

Inductive step: Assume $a^n b \leq x$ for some $n \geq 0$. We need to show $a^{n+1}b \leq x$.

$$a^{n+1}b = a \cdot a^n b \leq a \cdot x \quad \text{(by inductive hypothesis)}$$

Since $a \cdot x \leq x$:

$$a^{n+1}b \leq x$$

By induction, $a^*b \leq x$. □

*5.2.4* $b + xa \leq x \implies ba^* \leq x$.

Proof. Assume $b + xa \leq x$. We need to show $ba^* \leq x$. Base case ($n = 0$):

$$ba^0 = b \cdot 1 = b \leq x \quad \text{(by assumption)}$$

Inductive step: Assume $ba^n \leq x$ for some $n \geq 0$. We need to show $ba^{n+1} \leq x$.

$$ba^{n+1} = b \cdot a \cdot a^n \leq x \cdot a \cdot a^n \quad \text{(by inductive hypothesis)}$$

Since $x \cdot a \leq x$:

$$ba^{n+1} \leq x$$

By induction, $ba^* \leq x$. □

## 5.3 Proofs for Parallel Composition

*5.3.1 Proof of Commutativity.*

Theorem 5.1. *For all programs $p$ and $q$, $p \parallel q = q \parallel p$.*

Proof. By definition, parallel composition is commutative:

$$p \parallel q \equiv \text{execute } p \text{ and } q \text{ in parallel} \equiv q \parallel p$$

□

*5.3.2 Proof of Associativity.*

Theorem 5.2. *For all programs $p, q, r$, $(p \parallel q) \parallel r = p \parallel (q \parallel r)$.*

Proof. By definition, parallel composition is associative:

$$(p \parallel q) \parallel r \equiv \text{execute } p, q, \text{ and } r \text{ in parallel} \equiv p \parallel (q \parallel r)$$

□

*5.3.3 Proof of Distributivity over +.*

Theorem 5.3. *For all programs $p, q, r$, $p \parallel (q+r) = (p \parallel q) + (p \parallel r)$.*

Proof. By definition of parallel composition and distributivity:

$$p \parallel (q+r) \equiv \text{execute } p \text{ in parallel with either } q \text{ or } r \equiv (p \parallel q) + (p \parallel r)$$

□

*5.3.4 Proof of Identity.*

Theorem 5.4. *For all programs $p$, $p \parallel 0 = p$.*

Proof. By definition of parallel composition and the identity element:

$$p \parallel 0 \equiv \text{execute } p \text{ in parallel with an empty program} \equiv p$$

□

## 5.4 Handling Undefined Behavior

Theorem 5.5. *If $p \parallel q$ results in a data race, it leads to an undefined state $\epsilon$.*

Proof. Assume $p \parallel q$ causes a data race. By definition of data races in parallel composition:

$$p \parallel q \text{ causes a data race} \implies \text{execution state} = \epsilon$$

This follows directly from the definition that a data race leads to an undefined behavior state $\epsilon$. □

# 6 APPLYING HARDKAT TO CALYX SEMANTICS

The following code represents a snippet of the translation object in the pipeline.

```
component main() -> () {
  cells {
    x = std_reg(32);
    y = std_reg(32);
  }
  wires {
    group g1 {
      x.in = y.out;
      x.write_en = 1'd1;
      g1[done] = x.done;
    }
    group g2 {
      y.in = x.out;
      y.write_en = 1'd1;
      g2[done] = y.done;
    }
  }
  control {
    par {
      seq {
        g1;
        @sync(1) g1;
```

```
      }
      seq {
        g2;
        @sync(1) g2;
      }
    }
  }
}
```

**Listing 11: Calyx Program with Synchronization**

## 6.1 Proofs with Observations in HardKAT

*6.1.1 (Proof) Handling Observations.* Consider an observation $a$ where $a ::= g_n = n \mid a \wedge b \mid a \vee b \mid \neg a$.

*6.1.2* $g_n = n$.

Proof. We need to show that the observation $g_n = n$ holds.

$$g_n = n \quad \text{(by definition of the global state observation)}$$

This observation states that the global state $g_n$ is equal to $n$. This is a direct definition and holds by construction of the observation. □

*6.1.3* $a \wedge b$.

Proof. Assume $a$ and $b$ are two observations. We need to show $a \wedge b$.

$$a \wedge b \quad \text{(by definition of logical conjunction)}$$

This observation states that both $a$ and $b$ hold simultaneously. This holds if and only if both individual observations are true. □

*6.1.4* $a \vee b$.

Proof. Assume $a$ and $b$ are two observations. We need to show $a \vee b$.

$$a \vee b \quad \text{(by definition of logical disjunction)}$$

This observation states that at least one of $a$ or $b$ holds. This holds if at least one of the individual observations is true. □

*6.1.5* $\neg a$.

Proof. Assume $a$ is an observation. We need to show $\neg a$.

$$\neg a \quad \text{(by definition of logical negation)}$$

This observation states that $a$ does not hold. This holds if and only if the observation $a$ is false. □

# 7 CONCLUSION AND FUTURE PROPOSAL

Overall, we showcase a cohesive solution to reasoning about symbolic execution algorithms in Calyx by implementing a translation between the Calyx sourcecode and the HardKAT IR. Because of time constraints, we were not able to do a formal test suite to compare the efficacy of the HardKAT semantics in identifying mistakes in the symbolic executor algorithm. However, we hope to show that the first snippet of a working front-end for a KAT framework that targets HDL. In addition, we also propose that in a future project, it would be possible to reason about complicated semantics, such as those showcased by `par`, using domain-specific KAT semantics, which would alleviate the stress on using a general framework for implementing front-ends in various compilers.

# REFERENCES

[1] Axiomatic concurrency in hardkat. https://www.youtube.com/watch?v=LgOyvv9ZRPU.
[2] Calyx Evaluation. https://github.com/cucapra/calyx-evaluation.