Diploma Thesis

A Framework for Modular and Compositional Reasoning in Kôika

Max Kurze

March 14th, 2025

Dresden University of Technology
Faculty of Computer Science
Chair of Compiler Construction

Supervising Professor:	Prof. Dr.-Ing. Jeronimo Castrillon
Supervisor:	Dr. Sebastian Ertel

Declaration

I hereby confirm that this diploma thesis is a work of my own, and that only cited sources have been used.

[illegible]

Dresden, March 14th, 2025

Max Kurze

Abstract

Ensuring the functional correctness of hardware circuits is essential for establishing trust. Formal verification methods such as model checking and assertion-based verification, while widely used, are inherently limited in expressivity and scalability. These limitations prevent them from bridging the semantic gap between low-level hardware implementations and high-level specifications, posing challenges for comprehensive verification on complex circuits.

This thesis addresses these challenges by developing a scalable proof infrastructure tailored for hardware verification. Specifically, it enhances the existing Kôika hardware description language by introducing an improved compiler frontend that facilitates the processing of parametric actions. Furthermore, it proposes a new proof infrastructure that formalizes Hoare logic within Kôika's semantics, enabling structured and modular reasoning about hardware behaviors.

These contributions advance the modular verification of hardware circuits, overcoming the scalability limitations of conventional verification approaches. Future research directions include extending this framework to incorporate separation logic, thereby addressing the frame problem, and evaluating its applicability to larger hardware descriptions, such as a RISC-V implementation.

Contents

1 Introduction

Hardware circuits are at the heart of every digital system. Therefore, ensuring their functional correctness is of utmost importance, as it establishes the foundation of trust in these systems. Nowadays, this trust becomes ever more important as digital systems are integrated into a growing spectrum of domains, including security- and safety-critical applications, such as implantable medical devices or automotive brake control systems. Under these demands for trust, manual code inspection or testing alone are insufficient. Instead, correctness must be rigorously established through mathematical reasoning and formal proof.

At the same time, however, circuits have grown significantly more complex over the past decades [Tsc+02], making them more vulnerable to bugs and presenting challenges to the scalability of traditional formal methods.

1.1 Limitations of Existing Approaches

Most existing verification tools were designed by minimizing manual effort to enhance their accessibility for hardware developers while saving time and budget during development. Nevertheless, such automation comes at the expense of expressiveness, rendering these systems unable to scale to the complexity of today's circuits.

A notable example concerning this limitation is *model checking*, a method in which the verification is fully automated. However, this approach relies on an exhaustive exploration of all model states, leading to exponential growth of computational time and ultimately restricting verification to heavily abstracted portions of the actual circuit. Even though most parts of this state space could be proven irrelevant with only little manual effort.

Although these systems are designed to minimize interference with developers, recent studies reveal that formal verification constitutes more than half of the total development time for a typical ASIC or FPGA system [Fos20]. Since developers must engage with these systems regardless, their design should facilitate better interaction, ultimately leveraging developer expertise to manage complex systems more effectively.

Another limitation is that the most used hardware description languages operate at the register-transfer level, defining circuits at a very low level of abstraction. Consequently, their associated verification systems also function at this low level, whereas specifications typically express expectations in a highly abstract manner. For example, the IEEE standard for floating-point arithmetic [19] requires that the rounding operation produces the "number closest to [...] the infinitely precise result." However, hardware implementations primarily deal with primitive components such as wires and logical connectives, without any notion of numerical value at all. As a result, formal reasoning is typically constrained to low-level circuit behavior.

To bridge this gap and enable the verification of complex circuits against high-level specifications, a customizable proof framework is essential — one that can introduce new abstractions and facilitate reasoning about them. Typically, such approaches also elevate the level of abstraction in the hardware description language. However, it is crucial to ensure that the language maintains sufficient control over the generated circuit and that the validity of proofs is preserved throughout the compilation process.

1.2 Objectives

In this thesis, I propose a framework for modular reasoning over complex hardware within the Rocq proof assistant [BC13]. Rocq serves as a robust foundation as it provides higher-order logic, which enables formal reasoning about complicated properties. Additionally, Rocq provides programmable proof automation which can be combined with manual inductive reasoning for guidance.

The presented framework is based on Kôika [Bou+20], a language for the design and verification of hardware circuits. This language itself is integrated in Rocq to facilitate reasoning about hardware descriptions. Kôika offers imperative commands which are accompanied by formal semantics to support mathematical reasoning regarding their evaluation.

The goal of this work is to enhance the existing Kôika infrastructure, enabling the description of parametric circuits and providing a robust proof framework for reasoning about them. Specifically, this work introduces (a) a new frontend capable of parsing and type-checking parametric designs and (b) a proof infrastructure that allows for reasoning about these parametric designs while supporting the modular composition of proofs to address the complexity of larger circuits.

1.3 Outline

This thesis begins with a comprehensive background on formal methods and hardware descriptions in Chapter 2. It then progresses to Chapter 3 and Chapter 4, which provide insights into the implementation of the proposed reasoning framework while also highlighting various challenges encountered during the process. In detail, Chapter 3 discusses the new frontend (a), while Chapter 4 focuses on a Hoare logic for modular reasoning (b). Following this, Chapter 5 evaluates the practical applicability of the framework through a series of case studies. Lastly, Chapter 6 summarizes this study's findings and suggests potential directions for future research.

2 Literature Review

Before delving into the implementation details of the proposed validation framework, it is essential to establish the necessary theoretical foundations. The objective of this chapter is to provide an overview of common approaches to hardware verification and to analyze their limitations. However, a basic understanding of hardware design is required to fully grasp the rationale behind these validation methodologies and their integration into the development process. Therefore, this chapter begins with an introduction to hardware descriptions before exploring various verification approaches in detail.

2.1 Hardware Description Languages

Hardware systems are typically designed and specified in *hardware description languages (HDLs)*. These languages facilitate different tasks including simulation, synthesis, and analysis at various levels of abstraction. For that, the system is conventionally modeled as a *finite state machine (FSM)*. These FSMs consist of states which represent hardware registers and transitions which are defined by the computational logic of the circuit.

Certain aspects, such as placement, referring to the physical positioning of components, and routing, which involves the precise design of interconnecting wires, are abstracted in most HDLs. Instead, the exact circuit layout is generated programmatically during a process known as *synthesis*. Furthermore, most HDLs primarily focus on digital logic representations. Since the circuits examined in this thesis are exclusively digital, the discussion does not extend to the specifics of analog hardware descriptions.

Depending on their philosophy and level of abstraction, there exist different types of hardware languages. The following is going to elaborate on their differences and their applicability for formal hardware verification.

2.1.1 Register-Transfer Languages

Among the earliest languages are *register transfer languages* (RTLs), which remain the most dominant type to this day. Two popular examples include Verilog [96] and VHDL [88], which both got published in the mid 1980s and still play an important role in today's hardware development. One of their primary advantages, but also a notable disadvantage, is their provision of low-level constructs for specifying behavior, timing, and connectivity. While this grants the designer significant control, it comes at the cost of highly verbose code.

However, as these languages are not designed with formal verification in mind, they only offer prose-based semantics. Although it is theoretically feasible to derive formal behavioral definitions from these specifications, this process is highly complex and prone to errors due to ambiguities [Mer+10]. Furthermore, their low level of abstraction significantly complicates the verification of high-level properties, making formal analysis particularly challenging.

2.1.2 Software-Inspired Approaches

To enable faster design explorations and to make hardware development more accessible, various approaches have attempted to adapt software languages for hardware design, including SystemC [06] and SpecC [Gaj+00]. In these languages, designs are typically expressed as sequential C or C++ programs and then synthesized into parallel circuits through extensive compiler analysis. Some approaches further augment these languages with explicit parallel constructs or impose restrictions to improve hardware synthesis.

While these techniques can successfully generate hardware from software-based descriptions, the fundamental semantic gap between the specification language and the resulting circuit often leads to unpredictable outcomes in terms of size and performance. Moreover, developers only gain limited control over these critical design metrics, making it challenging to optimize hardware implementations effectively.

One advantage of these languages is their level of abstraction, which facilitates reasoning about micro-architectural properties. However, similar to Verilog and VHDL, these software languages lack formal semantics, hindering formal verification [Var07]. Moreover, a more critical limitation is that their compilers are not formally verified to preserve the semantics of the high-level language, thereby compromising the reliability of proofs conducted at this level of abstraction. Furthermore, due to the unpredictable impact of their high-level constructs on the generated circuit, it is virtually impossible to offer guarantees regarding circuit area or performance. Nevertheless, different works have proposed model-checking based formal verification for the SystemC language [CNR13, GLD10, HPG15].

2.1.3 Guarded Atomic Action

An alternative approach aimed at increasing the abstraction level of RTL is known as *guarded atomic actions (GAAs)*. This methodology separates a hardware design into *structural* and *behavioral* components. The structural components correspond to variables representing hardware registers, while the behavioral components are defined by *rules*. Each rule comprises an *action*, which specifies a state transition, and a *guard* condition. The guard ensures that the associated action can only be executed when the specified condition is satisfied.

The key idea behind this concept is, that whenever an action is executed, its transition is guaranteed to appear atomic. In other words, even if multiple actions are scheduled simultaneously, the system ensures that the overall state transition is equivalent to some sequential interleaving of those actions. This guarantee allows developers to construct hardware systems using a set of independent actions while ensuring the absence of unintended data races. As a result, it facilitates a more modular design, while preventing bugs from shared state. This concept is often referred to as *one-rule-at-a-time (ORAAT)* semantics.

The concept of atomic actions has its origins in early formalizations of concurrent processes. One of the foundational works in this area is C. A. R. Hoare's Communicating Sequential Processes [Hoa78], which introduces a language for parallel composition of sequential programs based on Dijkstra's Guarded Commands [Dij75]. Over time, several atomic action-based languages have been proposed taking these ideas from software design to hardware specifications, including Staunstrup's Synchronized Transitions [SG88], Dill's Murphi [Dil96], and Augustsson's Bluespec SystemVerilog [Nik04].

In the context of formal verification, GAA systems offer the advantage of a higher level of abstraction, similar to the previously discussed software-inspired languages. However, unlike these approaches, GAAs provide greater control over the generated hardware. Nevertheless, since these languages also rely on a compilation process, it is crucial to ensure that this process is formally verified to preserve the high-level semantics and thereby also the validity of proofs.

2.2 Formal Verification

Formal verification summarizes a range of mathematical techniques employed to rigorously establish the correctness of a system. However, for such verification to be meaningful, the notion of system *correctness* must be precisely defined. Such functional requirements are typically outlined in a *specification*, which is often expressed in natural language supplemented with pseudocode examples. Unfortunately however, a prose-based specification is insufficient for formal proofs, as it lacks the precision and rigor necessary for mathematical reasoning [Arm+19]. To ensure compatibility with formal methods, these specifications need to be formulated within the same logical framework as the proof itself or be at least convertible into it.

In contrast to conventional testing approaches, which assess a system based on a limited set of carefully selected scenarios, formal verification ensures correctness across all possible states using mathematical reasoning. As a result, formal verification provides stronger correctness guarantees, making it essential for safety-critical domains. However, since formal methods often require significant manual effort, they are best utilized in combination with conventional testing methods to achieve an efficient validation process.

Since most formal methods rely on a foundational logic for property specifications, the following provides a concise summary of propositional logic, first-order logic, and temporal logic.

2.3 Foundational logics

Propositional Logic. A proposition is a statement which is either true or false, while propositional logic establishes a mathematical framework for reasoning about such statements. In this system, statements are typically represented by single letters, referred to as *variables*. These variables, when combined with logical *connectives*, can be structured into more complex propositional formulas. Definition 1 shows a formal specification of these formulas for a frequently used subset of connectives.

$$\begin{array}{rcll}
X & \in & \mathcal{X} & \textit{set of variables} \\
A & := & \top & \textit{truth} \\
 & \mid & X & \textit{variable} \\
 & \mid & A \land A & \textit{conjunction} \\
 & \mid & \neg A & \textit{negation} \\
\bot & := & \neg \top & \textit{falsity} \\
A_1 \lor A_2 & := & \neg(\neg A_1 \land \neg A_2) & \textit{disjunction} \\
A_1 \rightarrow A_2 & := & (\neg A_1 \lor A_2) & \textit{implication}
\end{array}$$

Definition 1: Syntax of propositional logic formulas

First-Order Logic. This logic system can be regarded as an extension of propositional logic. As illustrated in Definition 2, it introduces predicates over objects, along with universal and existential quantification. In a first-order formula, the truth value of a predicate may depend on the specific objects to which it is applied. These objects, in turn, are not required to be logical entities. For instance, consider the predicate of equality. It is intuitively evident that the statement $2 = 2$ holds true, whereas $1 = 3$ is false. Similarly, it follows that objects such as 2 or 3 do not inherently possess a truth value and, thus, can only be used together with a predicate.

$$
\begin{array}{rcll}
A & \coloneqq & \top & \textit{truth} \\
 & | & X & \textit{variable} \\
 & | & A \wedge A & \textit{conjunction} \\
 & | & \neg A & \textit{negation} \\
 & | & P(t, ..., t) & \textit{predicate} \\
 & | & \forall x.A & \textit{universal quantification} \\
 & | & \exists x.A & \textit{existential quantification}
\end{array}
$$

Definition 2: Syntax additions of first-order logic

Linear Temporal Logic. Linear temporal logic (LTL) [Pnu77] can similarly be understood as an extension of propositional logic, albeit with a distinct purpose. As the name implies, LTL introduces operators that enable the formulation of properties over time. In this context, time is abstracted into discrete steps, and formulas are evaluated over infinite sequences of these steps. Definition 3 presents typical temporal operators in LTL, including the *until* operator U, which asserts that a property holds until another property becomes true, and the *next* operator $\bigcirc$, which indicates that a property holds in the immediately following time step. These fundamental operators can also be used to express that a property *always* or *eventually* holds.

$$
\begin{array}{rcll}
A & \coloneqq & \top & \textit{truth} \\
 & | & X & \textit{variable} \\
 & | & A \wedge A & \textit{conjunction} \\
 & | & \neg A & \textit{negation} \\
 & | & \bigcirc A & \textit{next} \\
 & | & A \,\mathsf{U}\, A & \textit{until} \\
\lozenge A & \coloneqq & \top \,\mathsf{U}\, A & \textit{eventually} \\
\square A & \coloneqq & \neg\lozenge\neg A & \textit{always}
\end{array}
$$

Definition 3: Syntax of linear temporal logic

2.3.1 Logic Notations

To facilitate a clearer understanding of the remainder of this work, it is important to briefly introduce the concept of inference rules. For that, Rule 1 illustrates the general structure of such a rule, which consists of multiple premises written above a horizontal line and a single conclusion below. Each rule asserts that whenever all premises are satisfied, the specified conclusion logically follows.

$$\frac{\mathit{premise}_1 \quad \dots \quad \mathit{premise}_n}{\mathit{conclusion}} \tag{1}$$

These rules are commonly used to inductively define relations or functions by specifying a rule for each possible input case, thereby establishing how inputs correspond to their respective outputs.

In the following sections, three different formal techniques, *assertion-based verification*, *model checking* and *theorem proving*, are examined in greater detail and their applicability for this study is evaluated.

2.3.2 Assertion-Based Verification

In an assertion-based verification approach, the hardware description of a circuit is enriched with assertions directly in the model [FKL03]. These assertions specify design expectations in form of properties over circuit components. For instance, an assertion may enforce that a register's value remains within a certain range. Depending on the toolchain used, assertion evaluation can occur dynamically during simulation, statically during synthesis, or even at runtime during the execution of the synthesized hardware [Das+06].

Example languages include Intel's Forspec [Arm+02] and SystemVerilog's assertions [05]. In SystemVerilog, these assertions are generally classified into *immediate* and *concurrent* assertions, depending on how they are integrated into the design. Immediate assertions are evaluated only at the moment they are encountered, whereas concurrent assertions operate independently of procedural execution. Consequently, concurrent assertions enable reasoning about temporal properties that extend across multiple clock cycles. These properties closely resemble those that can be expressed using LTL.

However, these properties remain fairly low-level, focusing on the states of individual wires. Due to the absence of powerful abstraction mechanisms, they are insufficient for expressing more complex properties related to the microarchitecture or the ISA specification.

2.3.3 Model Checking

Model Checking is a fully automated technique that verifies system properties by systematic, but exhaustive exploration of its state space. It was introduced in the early 1980s by Clarke and Emerson [CE81], and independently by Queille and Sifakis [QS82].

The process involves expressing the specification in terms of temporal logic properties, e.g., LTL properties. These properties allow for the verification of *liveness* properties which ensure that the system makes progress, as well as *safety* properties which ensure the absence of undesired behavior. For the actual verification process, the model checker constructs a finite-state graph representing all possible system states. The precise verification technique varies depending on the type of logic used to express the properties. In the case of LTL model checking, for example, the proposition is transformed into a state graph, and a path analysis is performed on the product system formed by combining the model with the proposition [BK08].

The biggest problem of model checking is the state space explosion. This refers to the exponential growth of the state space with regard to the system's size. In particular, every additional input bit or register bit in the system doubles the size of its corresponding model. Thus, in practice, systems need to be drastically reduced and abstracted in order to keep the computational effort feasible.

2.3.4 Theorem Proving

The approach of *interactive theorem proving (ITP)* is based on so called *proof assistants*. These proof assistants can be understood as computer programs which validate the correctness of a mathematical proof. This is done by defining all constructs in terms of a very small but powerful core calculus, typically based on type theory. By doing so, manually written proofs can be automatically verified by checking the correctness of their construction in this calculus.

This approach is a very general one as it is not restricted to the verification of hardware or software systems. Quite the opposite, it can be used to formulate and reason over arbitrary mathematical constructs. As a consequence, this approach is very expressive and could be used to model all kinds of systems and properties. However, for the same reason there is no single language of choice for modeling a system and therefore also no built-in proof automation for properties over hardware descriptions.

One prominent proof assistant utilized throughout this work is Rocq [CH85, CH88]. Initially developed in the 1980s, Rocq has since been the focus of extensive research and continuous development. It is based on the calculus of inductive constructions, which offers high expressivity for formulating mathematical assertions and constructing formal proofs. Furthermore, it includes a meta-language that facilitate customizable proof automation.

Early research already recognized higher-order logic as a suitable foundation for the formalization and verification of hardware designs due to its expressiveness and rigorous semantics [Gor85]. As a result, various studies have proposed HDLs based on type theory [CJ04, HDL90]. However, since these languages still describe circuits at the RTL level, their usability remains limited, and verifying their alignment with high-level specifications is challenging.

2.4 Kôika

In 2020 Bourgeat et al. introduced the Kôika language, combining the formal benefits of a language based on type theory with the abstraction gained from GAAs [Bou+20]. It is designed to offer rigorous formal semantics for reasoning about hardware descriptions. In addition, Kôika features a certified compiler which ensures functional correctness is maintained during the translation to Verilog. The compiler correctness is crucial as it guarantees that the validity of formal proofs about the high-level design is preserved throughout the compilation. An overview of this compilation process can be seen in Figure 1.

2.4.1 Compilation Process

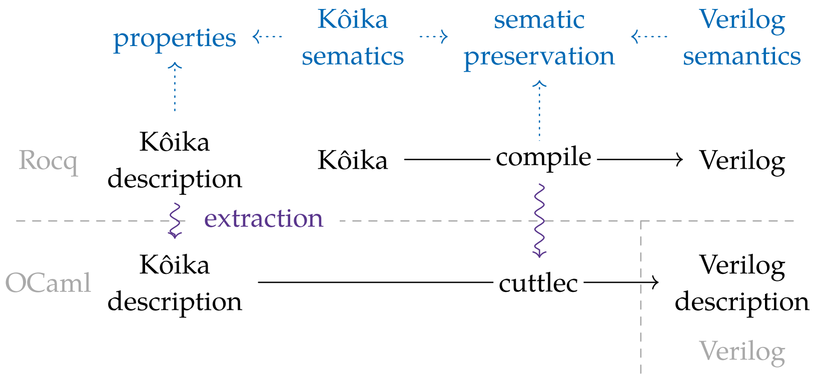

Figure 1: Kôika compilation overview

In a standard Kôika workflow, properties are formulated and formally verified against a circuit's implementation within Rocq. Subsequently, in order to generate a hardware circuit, both the Kôika description and the compiler itself must be extracted into OCaml. This extraction process is essential, as Rocq lacks a system interface for interacting with communication channels or performing file system operations. OCaml, on the other hand, is a functional programming language rather than a theorem prover like Rocq and therefore provides these standard facilities. However, since OCaml's type system is less expressive than that of Rocq, proofs are discarded during extraction. This does not pose an issue, as these proofs are solely relevant for verification and do not influence the program's execution. Finally, the Verilog description is generated by executing the extracted compiler, referred to as cuttlec, on the hardware description.

2.4.2 Kôika's Semantics

The semantics of the Kôika language are significantly influenced by prior research on GAAs, particularly Bluespec SystemVerilog. However, unlike these approaches, Kôika does not automatically infer the schedule of actions. Instead, it requires the developer to explicitly define their execution order. This design choice provides greater control and confidence in the resulting circuit behavior. However, it also enables the construction of schedules which contain actions with conflicting register accesses.

Consequently, to ensure consistency, Kôika needs to dynamically maintain a read/write log for each combination of registers and actions. These logs record all register interactions and are utilized to detect conflicts after an action's execution. In the event of a conflict, the later action in the schedule is *aborted*, meaning its log is discarded and never applied to the register state. Such a conflict arises, for instance, when two rules attempt to write to the same register in the same cycle, as Kôika only permits a single write per register per cycle.

2.4.3 Kôika's Verification

Kôika's compiler is implemented within the Rocq proof assistant, alongside the Kôika language itself, which is designed as an embedded language. As a result, the extensive Rocq ecosystem can be utilized to formalize and verify both the compiler and the hardware designed using Kôika. This ecosystem makes Kôika particularly well-suited for formal verification, as it enables the formulation and proof of complex high-level properties. Furthermore, the Kôika parser is also constructed using Rocq's extensible notation system, proving a user-friendly and intuitive input syntax without requiring external parsing tools.

As mentioned in Section 2.1.1, there exist no official formal semantics for Verilog. Thus, to prove the correctness of their compiler Bourgeat et al. first had to define own semantics for the Verilog subset which their compiler generates. These semantics, as well as the high-level semantics of Kôika, are based on various high-level constructs formalized in Rocq, including an environment for register state, a bit vector implementation, and a local context for the state of program variables.

2.5 Summary

In conclusion, this chapter identified two primary criteria for assessing the suitability of an HDL for formal verification. First and foremost, the language must have formally defined execution semantics, as formal reasoning is impossible without them. Moreover, retroactively defining these semantics is often excessively complex and prone to ambiguity. Secondly, the language should provide robust abstraction mechanisms to manage complex designs and facilitate the specification of properties close to the specification level.

Most of the languages discussed, which are commonly used in practical applications, lack either one or both of these requirements, presenting challenges for their integration with formal methods.

With regard to formal verification, the presented approaches demonstrated the trade-off between automation and expressiveness. The more complicated their properties may be the less automation can be expected.

Ultimately, this work aims to contribute towards a larger vision of enabling the verification of hardware implementations against their specifications. Consequently, the highest possible expressiveness needs to be achieved. For that, the formal approach of theorem proving is chosen together with the Kôika language. This language however, still lacks in two regards, which are addressed in this thesis.

1. Kôika descriptions cannot be parameterized, limiting their abstraction.
2. Proofs cannot be modularized, hindering the verification of complex properties.

In the following, Chapter 3 is going to address the first issue, while Chapter 4 is concerned with the solution of the second problem.

3 Typed Parsing for Parametric Designs

One of the main motivations for choosing theorem proving as an approach for hardware verification is the expressiveness it provides. The Kôika language, however, is still lacking in this regard. One of the main issues with Kôika is that it cannot type check parametric hardware descriptions. These are descriptions that refer to unspecified design variables, which are crucial for the abstraction of the implementation as well as for the generalization of proofs. In fact, Kôika proofs can only reason over typed actions, as the evaluation semantics are defined exclusively for them. Consequently, typing of parametric descriptions is essential for reasoning about abstract designs. Examples of such designs include n-bit adder circuits, FIFOs of arbitrary data size, or even an entire router network with an unknown 2D structure.

In order to overcome this limitation, different approaches have been evaluated. The following is first going to explain the details of Kôika's original parsing implementation to give an impression where this limitation originated. This knowledge is then used to develop different approaches for its solution. In detail, two solutions are proposed, the first one, presented in Section 3.3, modifies the existing type checker, while the second, explained in Section 3.4, builds a completely new language frontend.

3.1 Previous Type Checking Implementation

To understand the type-checking algorithm of Kôika, it is necessary to take a step back and get an overview of the whole frontend as visualized in Figure 2. This frontend consist of two main components, the *parser* and the *type checker*. Actually, there also exists a third component called the *lexer*, which operates even before the parser, turning the input string into a sequence of tokens. This conversion, however, is rather irrelevant as it is almost completely abstracted away by the utilized parser library. Thus, it is only mentioned here for completeness. The tokens are then processed by the parser, which builds a tree structure called *abstract syntax tree (AST)* depending on the precedence levels of the language constructs. Up until this point, no semantic analysis has happened, only the syntax of the input has been validated. Following the parsing, the type checker then traverses the AST to validate the language's semantics. Typically, the type checker is concerned with verifying that arguments produce the correct type of value for the place where they are used, but also properties like variable scoping are validated during the AST traversal.

$$\begin{tikzcd}
\text{Kôika description} \arrow[r, "\text{parsing}"] & \text{untyped AST} \arrow[r, "\text{typing}"] & \text{typed AST}
\end{tikzcd}$$

Figure 2: Kôika frontend overview

As Kôika is an embedded language in Rocq, it uses Rocq's extensible notation system to create a custom parser. This parser produces untyped ASTs, an example of which can be seen in Listing 1. Additionally, the example demonstrates that the AST does not contain any information about the type of `a` or `b` at this point. In fact, even if both would have conflicting types like `bits_t 4` and `bits_t 7`, the parser would still generate the same AST. To be precise, this information is actually part of the AST, but it is only stored in the function node which is higher up in the hierarchy than the shown excerpt.

```
fun min (a: bits_t 4) (b: bits_t 4)
            : bits_t 4 =>
        if a < b then a else b
```

$\xrightarrow{\text{parsing}}$

```
UIf (UBinop (UBits2 (UCompare cLt))
                   (UVar "a")
                   (UVar "b"))
               (UVar "a")
               (UVar "b")
```

Listing 1: A simple minimum function and its AST

To make sure that the implementation is actually typed correctly, the Kôika compiler uses a type-checking function which recursively traverses the AST. This function assigns a type to each node and simultaneously checks if all assigned types line up. In case the function succeeds, it returns a very similar AST which is now annotated with types, as can be seen in Listing 2. As shown, all variables are assigned with their type, note that even the binary comparison is assigned the bit width of its operands. However, in case this type checking fails it returns an error and the whole compilation aborts.

```
If (Binop (Bits2 (Compare cLt 4))
          (Var (... ("a", bits_t 4) ... ))
          (Var (... ("b", bits_t 4) ... )))
   (Var (... ("a", bits_t 4) ... ))
   (Var (... ("b", bits_t 4) ... ))
```

Listing 2: Typed AST of the minimum function

3.2 Shortcomings in the Type Checker

The problem with this approach arises when type checking a parametric function like the adapted version of `min` shown in Listing 3. Note how this description differs from Listing 1 by generalizing over the bit sizes of a and b using the parameter sz. When processing this description, the type checking would neither result in a success nor in a precise failure. It results in an unfinished state unable to make progress reporting "Tactic failure: Unexpected type checker output" with a blown up error message of thousands of lines.

```
fun min (a: bits_t sz) (b: bits_t sz) : bits_t sz =>
                if a < b then a else b
```

Listing 3: A generalized minimum function

Upon closer examination of the error message, it becomes evident that the type checker failed to make progress when computing the term eq_dec sz sz. Instead, it resorted to exhaustively unfolding and computing all possible terms, ultimately resulting in an excessively large error message.

This issue arises from the type checker's implementation as a function, which relies on an equality decision procedure to programmatically determine whether two given types are equal. This procedure, eq_dec, needs to be implemented separately for every Kôika type. In case of the min function, for instance, the type checker needs to compute the equality of two bit vector types bits_t n and bits_t m. This decision is, in turn, reduced to checking the equality of the natural numbers n and m. A condensed version of this decision procedure for natural numbers, with certain details omitted, is presented in Listing 4.

```
Fixpoint eq_dec (n m : nat) : {n = m} + {n <> m} :=
    match n, m with
    | O   , O   => left eq_refl
    | S _ , O   => right ...
    | O   , S _ => right ...
    | S n', S m' => ... eq_dec n' m' ...
    end.
```

Listing 4: Decidable equality for natural numbers

To better understand its functionality, natural numbers should be considered in terms of their internal representation, where each number is encoded as a sequence of successors of zero. For example, the number 4 is represented as the forth successor of zero: S (S (S (S O))). The decision procedure then takes two arguments, n and m, and determines their equality based on their outermost constructors. If both are O, their equality is trivially established by reflexivity. If one is a successor while the other is O, they cannot be equal, and a proof of their inequality is provided. In the final case, where both are successors, the procedure is applied recursively to establish their equality based on the equality of their predecessors.

When applying the procedure to concrete values from the description in Listing 1, the equality of the types `bits_t 4` and `bits_t 4` can be easily determined by the `eq_dec` function. A corresponding call graph illustrating this invocation is presented in Figure 3.

```
eq_dec (S (S (S (S 0)))) (S (S (S (S 0))))
eq_dec    (S (S (S 0)))     (S (S (S 0)))
eq_dec       (S (S 0))         (S (S 0))
eq_dec          (S 0)             (S 0)
eq_dec             0                 0
eq_refl
```

Figure 3: Kôika frontend overview

However, when type checking the general description from Listing 3, the type checker lacks knowledge of the concrete value of `sz`. Although both types refer to the exact same variable and must therefore hold the same value, this information is not accessible to the type-checking algorithm. Moreover, the algorithm cannot pattern-match on `sz`, as it is unknown whether it is a successor or zero. Consequently, the existing type-checking implementation fails to make progress at this stage. It does not produce an AST, even though the types are correct. In fact, a corresponding typed AST could be constructed manually by providing a special proof of equality.

Thus, an alternative approach is necessary to circumvent this limitation, by solving these type equalities in an environment where the equality of variable references can be leveraged.

3.3 Ltac Approach

One of these environments is the context of *Ltac* scripts. Ltac is a meta-programming language integrated in the Rocq proof assistant [Del00]. It is typically used to automate the construction of proofs through the creation of specialized *tactics*. These tactics are algorithms executed in an environment where they can manipulate the proof goal and its hypothesis, or search for specific patterns and apply theorems.

As a consequence, the situation in which two types refer to the same variable can be identified through a specific pattern and addressed with a dedicated proof. Specifically, this pattern takes the form `eq_dec ?n ?n`, where `?n` represents a placeholder for an arbitrary term. To ensure the pattern matches correctly, the previous strategy — which greedily unfolded every term — had to be replaced. Otherwise, the unfolding of the identifier `eq_dec` into its definition would have prevented the pattern from matching. Instead, the combined tactic "`with_strategy opaque [eq_dec] cbn`" has been employed. In this tactic, `cbn` performs a quite similar evaluation, while the former strategy modifier makes sure that `eq_dec` is not unfolded by marking it `opaque`. Unfortunately however, this tactic is less efficient than the original one, resulting in slower type checking. Thus, it is integrated as a fallback and only activated in case the original tactic fails.

Following this approach, it is possible to build a term representing the type-checked version of the `min` function. This term, however, is not the typed AST itself. Instead, it consists of an application of the type-checking function to the untyped AST together with a proof that this application results in a success. In fact, the same kind of term is generated when successfully type checking a non-parametric description with the original implementation.

Unfortunately, this term proves impractical for use in formal reasoning. A primary challenge stems from the necessity to reason about the structure of the hardware description, which requires computing the typed AST. However, evaluating this term to derive the typed AST is inherently leading to an excessively verbose representation. Function calls are fully inlined, and struct and enum definitions are entirely expanded within each typing expression. For complex circuits, this results in huge terms, significantly impeding the performance of the interactive proof infrastructure. Moreover, this inlining hinders the modular composition of formal proofs by destroying the structure of the AST. Lastly, the evaluation itself might lead to excessive computational overhead, due to the exhaustive nature of these tactics.

3.4 Typed Parsing

Since merely adapting the previous type-checking implementation proved insufficient, a fundamentally new approach was required. Ideally, this new approach should simultaneously address both challenges: typing of parametric descriptions and obtaining the typed AST.

Fortunately, such an approach was already outlined by the original authors of Kôika in an experience report [BP21]. However, their implementation had not been publicly available until recently. Consequently, this work independently adopted and developed the same methodology, going further with the implementation and addressing additional challenges that were not explored in the original proposal.

To overcome the problem of computing the typed AST from the untyped one, this approach skips the intermediate step of the untyped AST altogether and instead directly produces a typed AST while parsing. However, for this to work, the type system of Kôika needs to be expressed in terms of Rocq's type system, such that every correctly Rocq-typed AST represents a correctly Kôika-typed hardware description. Languages embedded this way are known as *intrinsically typed* [Bac+17, Ben+12].

In this manner, the second issue related to parametric designs is also resolved, as Rocq's type checker is indeed sufficiently powerful to directly unify identical variable references without computing their values. However, to leverage these capabilities, it is necessary to invoke Rocq's type checker directly on the typed AST. Since type checking occurs immediately after parsing, a new parser was required, which directly emits typed syntax elements.

For a better understanding of the difficulties of building a new parser, the next section is first going to give an overview of the Kôika syntax and the parsing framework before going into the implementation details.

3.4.1 Kôika's Syntax

From a high-level perspective each Kôika program consists of a set of actions along with a schedule that defines their execution order. However, the composition of these components into a Kôika program is entirely handled within Rocq, making it irrelevant for Kôika's parser. This parser is solely responsible for processing individual actions in isolation. The constructs available for defining such actions are formally specified in Definition 4.

Each of these constructs corresponds to a rule in the parser and is represented as a node in the final AST. As Kôika is implemented within Rocq, it leverages Rocq's extensible parser to define these rules.

3.4.2 Rocq's Extensible Parser

Rocq allows extending its own parser through a series of commands. These extensions, referred to as *notations*, are expanded into their definitions while parsing. Thus, as they are expanded before type checking, they are similar to macros in other languages. However, this notation system is sufficiently powerful to support the definition of a whole embedded language within it.

As an example, the original parsing rule of the sequence construct is presented in Listing 5. In this rule, the left-hand side defines a pattern to be

Actions	a	:= x	*variable*
		| c	*constant*
		| a ; a	*sequence*
		| let x := a in a	*binding*
		| x := a	*assigment*
		| if a then a else a	*branching*
		| read$_p$(r)	*reading*
		| write$_p$(r, a)	*writing*
		| $\circ_1$ a	*unary op.*
		| a $\circ_2$ a	*binary op.*
		| a [a]	*bit select*
		| f(a, ..., a)	*int. fun.*
		| extcall f(a)	*ext. fun.*
		| struct sig { (n := a)* }	*struct init.*
		| fail	*abort*
Ports	p	:= 0 | 1	

Unary Op.	$\circ_1$	:= !
Binary Op.	$\circ_2$	:= || | && | ^ | != | ==
		| < | <= | > | >=
		| <s | <s= | >s | >s=
		| ++ | << | >> | >>>
		| + | - | *
Constants	c	:= 0b(0|1)$^+$ | 0o(0-7)$^+$
		| 0d(0-9)$^+$
		| 0x(0-9|a-f)$^+$

Definition 4: Formal syntax of Kôika

matched, while the right-hand side specifies the corresponding term to which it is expanded. Within the pattern, `a` and `b` are recursively parsed as actions, whereas the semicolon is treated as concrete syntax due to its placement within single quotes.

```
Notation "a ';' b" := (USeq a b)
```

Listing 5: Untyped sequence parsing rule

3.4.3 New Parser Implementation

Given the existence of both typed and untyped AST nodes, an initial strategy for implementing typed parsing could involve substituting all untyped constructs with their typed counterparts within the parsing rules. After this transformation, the new notations closely resemble their original counterparts, as illustrated by the typed parsing rule for the sequence construct in Listing 6. In this rule the only difference is the substitution of `USeq` with `Seq`.

```
Notation "a ';' b" := (Seq a b)
```

Listing 6: Typed sequence parsing rule

This approach proves effective for most small and simple actions. However, when these notations are nested to build more complex descriptions, Rocq's type checker occasionally fails to verify the correctness of their types. A minimal failing example is presented below in Listing 7, where a value is first read from the register `reg` and then a single bit of this value is selected.

```
read₀(reg)[0b1]
```

Listing 7: Action with typing problems

For this action Rocq's type checker produces a warning indicating that it was unable to confirm whether a single bit suffices for the selection. For instance, if the register had a size of 4 bits, selecting a specific bit would already require a 2-bit index. The underlying issue here, is that Rocq is unable to infer the size of the register, on which the index size depends. To understand why this problem occurs and how it needs to be solved, a deeper analysis of Rocq's and Kôika's type checking is necessary.

3.4.4 Typing Issues

In Rocq, type checking a term involves recursively determining the types of all its arguments before unifying the resulting type of the term with its expectation. This expectation might be an explicit type annotation, or a parameter type if the term itself is used as an argument within another expression. As a consequence, type checking operates bottom-up, i.e., types of inner expressions are determined first, before the overall type of the term is unified.

Since Kôika ASTs are constructed using Rocq terms, their type checking follows the same order. However, Kôika employs complex types to enforce the correctness of its ASTs. For instance, each AST node carries a list in its type, which specifies the set of local variables in scope, together with their type. This set, referred to as *variable signature*, must be referenced by variable identifiers to ensure their validity within the current scope.

Listing 8 illustrates this standard type-checking process on the example of a `Bind` construct, representing the AST node of "**`let`** `x := a₁ in a₂`". In this example, the signature is simplified to include only variable names. For this term the first argument `a₁` is typed first. However, as indicated by the meta-variable `?sig`, the signature remains unknown at this stage. The same applies to the second argument `a₂`, where only the newly bound variable `"x"` is known and appended to `?sig`. Consequently, the validity and type of references to `"d"` or `"b"` cannot be established while typing these actions. The meta-variable `?sig` is only resolved once the entire term has been processed and its type is unified with the expected type `action ["d";"b"]`.

$$\underbrace{\texttt{Bind "x" } \overbrace{\texttt{a}_1}^{\texttt{action ?sig}} \; \overbrace{\texttt{a}_2}^{\texttt{action ("x"::?sig)}}}_{\texttt{action ["d";"b"]}}$$

Listing 8: Expected types of a `Bind` term

Generalizing this example, the issue with the variable signature is that it needs to be constructed top-down, which contrasts with the standard parsing order. This discrepancy stems from the fact that the signature is derived from the structure of the AST. Typically, the root construct of the AST is initialized with an empty signature. This signature must then be propagated downward to ensure that binding constructs transmit their bound identifiers into nested actions.

Similar issues arise with the register typing function, which assigns a type to each register, as well as with the typing function of external procedures. These functions are likewise specified on the top-level type annotation of a description and must be forwarded into nested actions to resolve the types of reads and writes, as well as the types of external function calls.

To address these issues, it was necessary to modify the order of Rocq's type checking, such that these meta-structures were propagated prior to typing the inner actions. Fortunately, Rocq provides *bidirectionality hints* which allow modifying the unification order. Using these hints, Rocq can be instructed to unify the resulting term type first, which effectively transfers the necessary structures into the typing context. An example of such a hint is shown in Listing 9, where the position of the ampersand `&` indicates when the unification of the term type should occur.

```
Arguments Binop & ... {R Sigma} {sig} fn a1 a2
```

Listing 9: Bidirectionality hint for the BinOp construct

This hint already enables the type checking of the action from Listing 7, which motivated this modification. In this action the bit selection is represented by a `BinOp` in the AST, which now correctly forwards the register typing function due to the bidirectionality hint. Thus, enabling the read construct to determine the size of the register. However, for completeness, i.e., to ensure that the necessary information is propagated to every node of the AST, each construct has to be modified using such a hint.

3.4.5 Return Type Unification

Unfortunately, employing these bidirectionality hints introduces a different issue. Ideally, certain parts of the term's type should be unified early, as discussed in the previous section, while others should follow the default behavior and be unified last, as for example the return type of an action. Rocq's bidirectionality hints however, do not provide such fine-grained control. Instead, the entire type has to be unified at once.

A minified example where this behavior results in a problem is presented through a simple bit vector concatenation in Listing 10. The issue arises due to the early unification of the return type, caused by the bidirectionality hint. As this unification occurs before the sizes of the bit vectors are determined, the return type of the concatenation is initially inferred as `bits_t (?n + ?m)`. However, the type annotation expects the type `bits_t 10`, leading to a unification failure because Rocq cannot directly align the addition expression with the concrete value `10`. This unification can only succeed after type checking the nested actions, which ultimately resolves the type to `bits_t (4 + 6)`. Only once these precise values are determined, Rocq can evaluate the addition and verify that its result matches `10`.

```
0b0000 ++ 0b000000 : bits_t 10
```

Listing 10: Return type unification problem

The solution proposed here is to postpone the unification of these types by introducing a wrapper function `delay_tau_uni` with an additional parameter, which is resolved last and serves as a proof of the type equivalence. This wrapper function is illustrated in Listing 11. It solves the issue by splitting the return type into to separate types `tau_in` and `tau_out`. This prevents Rocq from attempting to unify these types prematurely. Instead, unification only occurs when the proof `H` is encountered. By this time however, the nested action `a` has already been fully typed, allowing the proof to follow naturally by reflexivity.

```
Definition delay_tau_uni {tau_in tau_out R Sigma sig}
  (a : action' R Sigma sig tau_in) (H: tau_in = tau_out)
  : action' R Sigma sig tau_out :=
    match H with
    | eq_refl => a
    end.
Arguments delay_tau_uni & ... {R Sigma sig} a H.
```

Listing 11: Function to delay the unification of the return type

To solve the bit vector example, this function could be wrapped around the concatenation, causing the unification of `tau_in` with `bits_t (?n + ?m)` and of `tau_out` with `bits_t 10`. Then, when reaching the proof `H`, `?n` and `?m` would be already resolved and their equality could be established trivially.

To systematically prevent these return type errors, the wrapper function should be applied wherever a specific return type is expected to delay the unification of the inferred type with this expectation. Such positions include the condition of an `If` construct, where a single bit is expected, or the second argument of a binary operation, which is frequently required to match the type of the first argument.

4 Hoare Logic for Modular Reasoning

To overcome the scalability problem of Kôika and enable proofs over more complex circuits, it was necessary to make these proofs more modular by decomposing them into smaller subproofs. Each smaller proof would then focus solely on reasoning about a specific part of a circuit.

For example, consider the proof goal that a program a_1 ; a_2, consisting of two parts, satisfies a property P. As formalized in Rule 2, a modular proof would establish this goal through two subproofs: one demonstrating that a_1 satisfies P_1, and another one showing that a_2 satisfies P_2. To achieve this, an algorithmic approach is required to transform the property P into the properties P_1 and P_2, ensuring that their composition logically implies P.

$$\frac{a_1 \vdash P_1 \quad a_2 \vdash P_2}{a_1 \; ; \; a_2 \vdash P} \quad assuming \quad P_1 \land P_2 \rightarrow P \tag{2}$$

This leads to the concept of *program logics*, which are formal systems designed for reasoning about the evaluation of language constructs. While the provided example focuses solely on the sequence construct, the underlying concept can be applied more generally, as demonstrated in the following sections. The remainder of this chapter is first going to give an overview of Hoare logic, a prominent example of a program logic in Section 4.1, before describing the details of this logic's implementation in Kôika.

4.1 Hoare Logic Background

In 1969, C. A. R. Hoare introduced a program logic for reasoning about functional program correctness, which later became known as Hoare Logic [Hoa69]. At the core of this logic system is the notion of a *Hoare triple*, which is used to formally express semantic properties over the execution of a program. Each Hoare triple is structured as follows:

$$\{P\}\, S\, \{Q\}$$

In this notation, P and Q are logical assertions about the machine's state, while S represents a program. In the original work of Hoare, these programs were command sequences of a simple imperative language, but subsequent works also developed rules for more complex constructs including recursive procedures [Hoa71] and concurrency [Hoa72].

The left assertion of the triple, denoted as P, is termed the *precondition*, as it specifies an initial hypothesis which is assumed to hold. Likewise, Q is referred to as the *postcondition*, representing the expected state after the execution of S. The Hoare triple as a whole asserts that whenever P holds in a given program state, then the execution of S will result in a state where Q is satisfied.

4.2 Hoare Triples for Kôika

Since Kôika already provides execution semantics, Hoare triples should be defined based on these semantics to ensure consistency. This approach offers the additional advantage that the Hoare rules introduced later can be formally proven correct with respect to these semantics, rather than being postulated as axioms, as originally done by Hoare.

In detail, Kôika's operational semantics are expressed through a big-step evaluation function, which directly computes the value of a given expression. This evaluation function operates based on a register environment $\mathcal{R}$, a context Γ, and two distinct logs — a scheduler log L and an action log l. The register environment stores the register values at the beginning of the cycle, while the context maintains the current values of local variables. The logs track prior reads and writes, with a crucial distinction, the action log only captures register accesses of the current action, while the scheduler log aggregates the accesses of all preceding actions in the schedule. This distinction is necessary to discard the accesses of the current action in case it is aborted. This evaluation function is formally expressed as follows:

$$\mathcal{R}, L \vdash (l, \Gamma, a) \downarrow (l', \Gamma', v)$$

In this notation, a denotes the action being evaluated and v its resulting value. Notably, this function only produces updated versions of the context and the action log, while the scheduler log and register environment remain uneffected by the evaluation process.

In the context of Hoare triples, the assertions P and Q must be capable of reasoning about the entire state of the circuit. As a result, they require access to the register environment, the variable context, and both logs. Kôika's Hoare triple definition then follows from Hoare's original one. It states that whenever P holds in a given state and the big-step evaluation function $\downarrow$ is applied to an action a in this state, then Q must hold in the resulting state. A formal version of this notation is shown in Definition 3. In this formula, the notation $C \vdash P$ should be interpreted as: property P is satisfied in the context C.

$$\{P\}\, a\, \{Q\} \quad := \quad \frac{\mathcal{R}, L, l, \Gamma \vdash P \quad \mathcal{R}, L \vdash (l, \Gamma, a) \downarrow (l', \Gamma', v)}{\mathcal{R}, L, l', \Gamma' \vdash Q(v)} \tag{3}$$

It should be noted that in this formulation the postcondition Q additionally depends on the computed value v. This dependency arises because every Kôika action is an expression that evaluates to a value, whereas the imperative language originally used by Hoare consisted solely of statements, which do not produce values. Consequently, from this point on, the postcondition of a Kôika Hoare triple is assumed to be an assertion that depends on the evaluation result. The notation $Q(v)$ is used to specify a particular value, while $\{v.\ Q\}$ binds the value to a name v, i.e., $\{v.\ Q(v)\}$ is equivalent to $\{Q\}$. The context variables $\mathcal{R}$, L, l, and Γ are similarly available within the environment enclosed by braces but named implicitly. They may also be specified explicitly for an assertion by the $\vdash$ notation, i.e., $\{P\}$ is equivalent to $\{\mathcal{R}, L, l, \Gamma \vdash P\}$.

Unfortunately, due to Kôika's logs, this definition turned out to be rather inconvenient to use. For example, to state that a register r holds a value v_r, one must consider prior writes to this register in the same rule ($\texttt{write}(r, v_r) \in l$), writes of prior rules ($\texttt{write}(r, v_r) \in L$) and the register valuation from the start of the cycle ($\Gamma[r] = v_r$). Ideally, it would be preferable to first commit the logs to $\mathcal{R}$, allowing reasoning to be conducted solely on this unified state.

However, to allow the combinations of the logs with the register environment, it is necessary to show that such a transformation would not impact the evaluation semantics. In detail, it needs to be shown that the evaluation on an environment where the logs have been committed into the register context results in the same value and updates as the normal evaluation. This statement can be seen formally in Rule 4. In this rule the notation $\mathcal{R} \oplus l$ represents the update of environment $\mathcal{R}$ with log l, while $+\!\!+$ denotes log concatenation and $[]$ an empty log.

$$\frac{\mathcal{R}, L \vdash (l, \Gamma, a) \downarrow (l +\!\!+ l', \Gamma', v)}{(\mathcal{R} \oplus L \oplus l), [] \vdash ([], \Gamma, a) \downarrow (l', \Gamma', v)}\ \textsc{LogIrr} \tag{4}$$

Assuming that Rule 4 holds, one might question the necessity of maintaining both logs instead of directly updating the register valuation. The rationale here is that this rule assumes that the evaluation succeeds in the first place. However, the determination of whether it succeeds cannot be made based solely on the register evaluation. Therefore, the logs are indeed necessary for the evaluation, and the rule only asserts that the precise output, in case the evaluation succeeds, could also be derived without logs. In the following, this rule is termed *log irrelevance*.

4.2.1 Achieving Log Irrelevance

The problem with this approach is that Rule 4 cannot be proven correct under the original execution semantics of Kôika. In detail, there is one special case which prevents this proof. To better understand this case, it is explained with an example. Consider an action log containing a write to a register $[\texttt{wr}(r, v_l), ...]$ and a register valuation R such that $R[r] = v_r$ and $v_l \neq v_r$. In this case, the evaluation of an action $\texttt{read}_0\texttt{(r)}$ would result in the value from the beginning of the cycle v_r. However, in case the logs and the valuation would have been collapsed, the only value kept associated to r would be the one of the last write v_l and there would be no way to recover the value v_r. In other words, it turns out that the log is not completely irrelevant for the result of the evaluation.

As a consequence, it became necessary to adapt the semantics of Kôika for the satisfiability of this rule. Specifically, the evaluation was constrained such that performing a $\texttt{read}_p$ after a $\texttt{write}_p$ would lead to an abort. With this modification, the log serves exclusively to determine whether an abort occurs. Furthermore, this adaptation is relatively minor, as reading from a register after writing to it was already considered poor coding practice in Kôika.

With the log irrelevance theorem established, a modified version of the Hoare triple definition can be constructed while preserving its original expressiveness. A formal definition can be seen in Rule 5.

$$\{P\}\, a\, \{Q\} := \frac{\mathfrak{R}, \Gamma \vdash P \qquad \dfrac{\mathcal{R}, L \vdash (l, \Gamma, a) \downarrow (l +\!\!+ l', \Gamma', v)}{\mathfrak{R}, [] \vdash ([], \Gamma, a) \downarrow (l', \Gamma', v)}\,\textsc{LogIrr}}{\mathfrak{R}', \Gamma' \vdash Q(v)} \begin{array}{l} \\ \text{with } \mathfrak{R} := \mathcal{R} \oplus L \oplus l \\ \text{and } \mathfrak{R}' := \mathcal{R} \oplus L \oplus l' \end{array} \qquad (5)$$

In this version the properties P and Q reason solely over the local variable valuations Γ/Γ' and the combined register valuations $\mathfrak{R}/\mathfrak{R}'$, thereby simplifying both assertions and proofs.

4.2.2 Stating Aborts

In classical Hoare logic, a Hoare triple always assumes that the command sequence terminates. Consequently, if a program fails to terminate, any conclusion can be derived from it. This enables the formalization of non-termination using an unsatisfiable postcondition. For example, the Hoare triple $\{P\}\, S\, \{\bot\}$ expresses that S does not terminate under the assumption P.

However, in contrast to most imperative languages, Kôika guarantees that descriptions always terminate. On the contrary, Kôika descriptions instead have the notion of aborts. Consequently, as the definition of Kôika Hoare triples assumes actions to succeed, the unsatisfiable postcondition can be used in Kôika to state that an evaluation aborts.

4.3 Hoare Rules

As Hoare triples are only used to formally state expectations about a program, additional infrastructure is necessary to actually reason over these statements and to construct formal proofs. For that purpose, Hoare logic provides a set of axioms and inferences rules. These rules are largely dependent on the language used, with at least one rule required for each language construct to ensure comprehensive reasoning. Specifically, each rule defines how a command's execution alters the program's state and how this, in turn, affects the pre- and postcondition.

4.3.1 Rules for Language Constructs

One prominent example rule of Hoare logic, is the one for variable assignment shown in Rule 6. This rule is actually rather an axiom as it does not contain any premises. The notation $P[E/x]$ denotes that every free occurrence of variable x in P is replaced by the expression E.

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\}}\ \textsc{Assign} \tag{6}$$

As an example, consider the postcondition $\{x = 5\}$ and assume x has been assigned the expression $y + 4$, then the required minimal precondition is $\{y + 4 = 5\}$.

Another fundamental Hoare rule worth mentioning is the sequencing rule as illustrated in Rule 7. This rule states that a sequence of commands can be decomposed by providing an intermediate assertion Q. Specifically, Q must follows from P through execution of the first sequence S_1 and must, in turn imply R upon execution of sequence S_2. This rule now essentially corresponds to the goal mentioned in the beginning of this chapter. It allows for the composition of a proof over a sequence construct from two smaller proofs.

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}\ \textsc{Seq} \tag{7}$$

4.3.2 Language-independent Rules

Furthermore, Hoare logic additionally includes language-independent rules to rewrite assertions as intermediate proof steps. These rules include one for substituting the precondition and another for the postcondition. The former is commonly referred to as *precondition strengthening*, as the most general precondition — i.e., $\top$ — is considered the weakest, since it is implied by the broadest set of assertions. Likewise, the latter is referred to as *postcondition weakening*, since, in contrast to the precondition, the postcondition is expected to be as specific as possible.

$$\frac{P \to P' \quad \{P'\}\, S\, \{Q\}}{\{P\}\, S\, \{Q\}}\,\textsc{StrenPre} \qquad\qquad \frac{\{P\}\, S\, \{Q'\} \quad Q' \to Q}{\{P\}\, S\, \{Q\}}\,\textsc{WeakPost}$$

Both rules incorporate the transitivity of the implication from classical logic into Hoare Logic. As a result, both rules serve a very similar purpose and are often combined into a single *rule of consequence* as shown in Conseq.

$$\frac{P \to P' \quad \{P'\}\, S\, \{Q'\} \quad Q' \to Q}{\{P\}\, S\, \{Q\}}\,\textsc{Conseq}$$

4.4 Hoare Rules for Kôika

To enable reasoning over Hoare triples in Kôika, it was necessary to implement a Hoare rule for each language construct. For that, it should be noted that there are multiple ways to define a Hoare rule for the same syntax element. At first, this may seem rather obvious as the precondition strengthening and postcondition weakening rules can be used to derive an infinite amount of less expressive Hoare rules. Take as an extreme example Rule 8, demonstrating the least expressive Hoare rule for an action consisting of a constant. However, even when considering only rules with maximal expressivity different versions exist. As an example, Rule 9, gives the most precise postcondition for an arbitrary precondition, while Rule 10, gives the most general precondition for an arbitrary postcondition. However, these rules serve a different purpose, one can be used for backwards reasoning while the other is rather suited for forwards reasoning. Consequently, to design a coherent system it is necessary to decide for one or the other approach and build all rules according to the same scheme.

$$\frac{c \text{ is a constant}}{\{\bot\}\, c\, \{\top\}}\ (8) \qquad \frac{c \text{ is a constant}}{\{P\}\, c\, \{v.v = c \land P\}}\ (9) \qquad \frac{c \text{ is a constant}}{\{Q(c)\}\, c\, \{Q\}}\ (10)$$

4.4.1 Predicate Transformers

Such a coherent Hoare rule system can be regarded as a *predicate transformer*. Originally introduced by E. Dijkstra [Dij75], a predicate transformer provides a systematic approach to derive the weakest precondition necessary for a given postcondition, or conversely, the strongest postcondition resulting from a given precondition. In this sense, predicate transformers perform a symbolic execution, translating the effects of statements into logical predicates. As a result, they reduce the verification of a Hoare triple $\{P\}\, S\, \{Q\}$ into the problem of proving the implication $P \rightarrow wp(Q, S)$, where $wp(Q, S)$ denotes the weakest precondition of S to satisfy the postcondition Q. It is important to emphasize that while this work technically implements Hoare triples rather than a dedicated predicate transformer function, it adheres closely to their underlying principles.

Since both, weakest precondition and strongest postcondition, are equally expressive, this work adopts the weakest precondition method, as backward reasoning aligns more naturally with the reasoning style of theorem provers. In detail, with this style, a proof always starts at a Hoare triple containing a desired postcondition and then progresses through the repetitive application of Hoare rules which match the current goal. These applications substitute the goal with a new one for each premise. Consequently, all rules must have an arbitrary postcondition in their conclusion and an arbitrary precondition in each premise. Intuitively, this criterion enables the composition of these rules, as the arbitrary pre- and postconditions can always be unified with their more specific counterpart.

The following section is going to present the implemented Hoare rules and explain their semantics.

4.4.2 Rules for Kôika

The rules Const, Var, and Read present the Hoare rules for the simplest Kôika constructs. Since these rules are designed for backward reasoning, they should be interpreted in that direction, beginning with the postcondition. In essence, these three constructs only produce a value without altering their environment. As a result, for an arbitrary assertion Q to hold in the postcondition, it must also hold in the precondition. However, because the precondition does not have direct access to the produced value, this value must be explicitly passed. For the Const rule this value can be directly inferred from the action, while Var and Read needs to retrieve it from the variable context and the register valuation respectively.

$$\frac{c \text{ is a constant}}{\{Q(c)\}\, c\, \{Q\}}\textsc{Const} \qquad \frac{x \in Var}{\{Q(\Gamma[x])\}\, x\, \{Q\}}\textsc{Var} \qquad \frac{r \in Reg}{\{Q(\mathfrak{R}[r])\}\, \texttt{read}_p\texttt{(}r\texttt{)}\, \{Q\}}\textsc{Read}$$

Another quite simple rule is the sequence rule, which closely resembles the original Hoare rule, with the only difference being that the result v of evaluating a_1 is explicitly discarded. This value holds no significance, as the return type of the first action in a sequence must be a bit vector of length zero. Consequently, the only possible value for v is the empty bit vector ε.

$$\frac{\{P\}\, a_1\, \{v.\ Q\} \quad \{Q\}\, a_2\, \{R\}}{\{P\}\, a_1 \texttt{ ; } a_2\, \{R\}}\textsc{Seq}$$

The IF rule follows a similar concept to the SEQ rule, with the difference that the result v of action a_c is not discarded but used to decide which postcondition should hold. In fact both rules could be considered special cases of a more general match rule which first evaluates a condition and then uses this value to decide between an arbitrary amount of postconditions, each being a precondition of its respective branch. For Kôika, however, such a general match rule is not necessary, as Kôika's match statement is only syntax sugar for nested if-then-else statements.

$$\frac{\{P\}\, a_c\, \{v.\ Q_t \text{ if } v = \texttt{0b1} \text{ otherwise } Q_f\} \quad \{Q_t\}\, a_t\, \{R\} \quad \{Q_f\}\, a_f\, \{R\}}{\{P\}\, \texttt{if}\ a_c\ \texttt{then}\ a_t\ \texttt{else}\ a_f\, \{R\}}\textsc{If}$$

More interestingly, the ASSIGN rule gives an example involving context manipulation. Reading this statment backwards again, it states that a postcondition of an assignment likewise needs to hold on the previous context Γ where the variable x has been assigned the value v explicitly, denoted as $\Gamma[x \mapsto v]$. Additionally, as the assignment is a statement with no return value, the empty bit vector ε is passed to Q directly. Just like VAR and READ were similar also WRITE closely follows ASSIGN. Their only difference is that WRITE modifies the register valuation $\mathfrak{R}$ instead of the variable context Γ.

$$\frac{\{P\}\, a\, \{v.\ \mathfrak{R}, \Gamma[x \mapsto v] \vdash Q(\varepsilon)\}}{\{P\}\, x \texttt{ := } a\, \{Q\}}\textsc{Assign} \qquad \frac{\{P\}\, a\, \{v.\ \mathfrak{R}[r \mapsto v], \Gamma \vdash Q(\varepsilon)\}}{\{P\}\, \texttt{write}_p\texttt{(}r, a\texttt{)}\, \{Q\}}\textsc{Write}$$

The Bind rule also modifies the context, adding a variable assignment. However, its effect is limited to the execution of an action a_2. Afterwards, the variable x is removed from the context again, all other variables are kept with their modifications from a_2.

$$\frac{\{P\}\, a_1\, \{v.\ \mathfrak{R}_1, \Gamma \oplus [x \mapsto v] \vdash Q\} \quad \{Q\}\, a_2\, \{\mathfrak{R}_2, \Gamma \setminus \{x\} \vdash R\} \quad x \in Var}{\{P\}\, \texttt{let}\ x \texttt{ := } a_1\ \texttt{in}\ a_2)\{R\}}\textsc{Bind}$$

For the binary and unary expressions of Kôika, their operands are evaluated to values and their effects are defined by *denotational semantics* represented by the double brackets ⟦...⟧. Such denotational semantics establish the meaning of operations by translating them into pre-existing and well-defined concepts. While this translation then depends on the exact operation used, the Hoare rule still holds for every binary operation. Thus, the ∘ symbol is used to denote an arbitrary binary or unary operation. Also note how BinOp uses a nested triple to extend the scope of v_1 to access both v_1 and v_2 in the nested postcondition.

$$\frac{\{P\}\, a_1\, \{v_1.\, Q \land \{Q\}\, a_2\, \{v_2.\, R(\llbracket v_1 \circ_2 v_2 \rrbracket)\}\}}{\{P\}\, a_1 \circ_2 a_2\, \{R\}}\textsc{BinOp} \qquad \frac{\{P\}\, a\, \{v.\, Q(\llbracket \circ_1 v \rrbracket)\}}{\{P\} \circ_1 a\, \{Q\}}\textsc{UnOp}$$

The same nesting technique is leveraged for the function call rule IntCall. In this rule, all arguments a_1 to a_n are first evaluated to their values v_1 to v_n, which are then used to build the local context of f. The function f itself has no access to the local context of its caller, only to the context built from the argument values. After the function's execution, the postcondition R must again hold on the context Γ_n returned by the last argument's evaluation, because the condition R conceptionally belongs to the caller. Additionally, similar to the Seq rule, each argument has a postcondition which matches the precondition of the next argument.

$$\frac{\{P_0\}\, a_1\, \left\{v_1.\, P_1 \land ..\{P_{n-1}\}\, a_n\, \left\{v_n.\, \left(\mathfrak{R}_n, \left[v_1, ..., v_n\right] \vdash P_n\right) \land \{P_n\}\, f\, \{\mathfrak{R}, \Gamma_n \vdash R\}\right\} ..\right\}}{\{P_0\}\, f(a_1, ..., a_n)\, \{R\}}\textsc{IntCall}$$

Lastly, the action `fail`, which explicitly causes a rule to abort, can satisfy any postcondition as dicussed in Section 4.2.2. This behavior is directly reflected by its associated Hoare rule.

$$\frac{}{\{\top\}\, \texttt{fail}\, \{Q\}}\textsc{Fail}$$

5 Evaluation

The evaluation is conducted in two parts. In the first part, the parsing capabilities of the new frontend are assessed. Then, in the second part, the framework as a whole is considered with focus shifted towards the Hoare proof infrastructure.

5.1 Typed Parsing

To examine the effectiveness of the new parser, it was tested as a drop-in replacement for different existing Kôika descriptions. The following will go over some instances from the official Kôika examples [PM22] as well as one from its standard library.

5.1.1 Typing Kôika's Examples

For most of the official examples, the parser worked without problems right from the start, including `collatz.v`, `conflicts.v`, `conflicts_modular.v`, `cosimulation.v`, `gcd_machine.v`, and `external_rule.v`. Notably, it was possible to abstract the queue implementation of one of them. Previously, this implementation had to be dependent on the exact size of the data stored. However, with the new parser it was possible to abstract over this type and even facilitate generalized proofs. Listing 12 shows the implementation where `bits_t 32` has been replaced with an abstract type `tau`. This enabled the verification of data-independent properties over this queue implementation. These properties were defined and proven using the Hoare infrastructure, and are therefore discussed in Section 5.2.2. It should be noted, however, that this queue is still rather basic as it can only store a single data item.

```
fun enqueue$_0$ (val: tau) : unit_t =>        fun dequeue$_1$ () : tau =>
  guard(read$_0$(empty));                       guard(!read$_1$(empty));
  write$_0$(empty, 0b0);                        write$_1$(empty, 0b1);
  write$_0$(data, val)                          read$_1$(data)
```

Listing 12: Generalization of queue implementation

5.1.2 Typing Kôika's Standard Library

Furthermore, the new parser was utilized to perform type checking on portions of Kôika's standard library. For instance, it successfully type checked the `valid` function shown in Listing 13, which was not possible previously due to its dependence on the parameter `tau`.

```
fun valid (x: tau) : struct_t Maybe =>
  struct_t Maybe::{ valid := 0b1; data := x }
```

Listing 13: A function of Kôika's std lib

Likewise, the entire FIFO implementation of the standard library was successfully type checked while maintaining it parametric over the type of data stored. This FIFO implementation bears some similarity to the queue depicted in Listing 12.

5.1.3 Design Flaws in the AST

An issue that emerged during these case studies exposed a design flaw in the type definitions of the AST nodes. Specifically, in the unary and binary operations, where the argument types and the return type are unified into a single one.

A simplified expression illustrating this issue is presented in Listing 14. In this example, the specific bit vector values are irrelevant, as only their types matter. Due to the AND connective, the left-hand side and the right-hand side are required to have the same type — more precisely, their return types must be equal. However, since their argument types are stored within the same data structure, they are unintentionally affected by this unification.

For the given example, the `retSig` function is intended to extract the last entry, yielding the type `bits_t 1` for both sides. However, because these types already match syntactically, the `retSig` function is never evaluated. Instead, Rocq's type checker directly unifies the meta-variable `?sz` with `1` to align both types. Unfortunately, this results in an unnecessary restriction, forcing the variables `a` and `b` to have the type `bits_t 1`.

$$\overbrace{\texttt{(0b1 || 0b1)}}^{\texttt{bits\_t (retSig \{\$ 1 \textasciitilde> 1 \textasciitilde> 1 \$\})}} \texttt{ \&\& } \overbrace{\texttt{(a > b)}}^{\texttt{bits\_t (retSig \{\$?sz \textasciitilde> ?sz \textasciitilde> 1 \$\})}}$$

Listing 14: Expression with incorrect unification

The only viable solution to this issue seems to be restructuring the types of the unary and binary operations. However, implementing this change would be time-consuming, as it interferes with large parts of Kôika's implementation and requires modifications throughout the codebase.

5.1.4 Escaped Functions

Another issue encountered relates to certain components of Kôika's standard library that make use of a special feature of the language to bypass the parser. This mechanism allows portions of the AST to be generated through custom Rocq functions. Such Kôika meta-programming facilitates the construction of complex macros by leveraging Rocq features, such as recursion, to systematically generate large and repetitive sections of the AST. An example of such a macro could be a switch tree that determines which register in a register file is read based on a selector bit vector.

The original type-checking implementation attempted to handle these macros by expanding them into their definitions. Through this approach, recursive macros were unrolled as far as possible. If the resulting structure formed a standard Kôika AST, the type checker could proceed as usual. However, if any macros remained that could not be fully expanded, type checking would ultimately fail.

Since these macros bypass the parser, actions that depend on them cannot be seamlessly adapted by merely switching to the new parser implementation. Instead, these macros must be manually re-implemented to directly generate typed ASTs. However, by re-implementing them, developers can take advantage of the typical benefits provided by the additional type information.

5.2 Hoare Infrastructure

To assess the usability and effectiveness of the proposed proof framework, it is evaluated on different case studies with increasing complexity.

5.2.1 Verifying Parametric Descriptions

The first case study evaluates the applicability of the Hoare framework for verifying parametric actions. To this end, a simple function from the `collatz` example was selected and generalized to bit vectors of an arbitrary size `sz`, as shown in Listing 15. Subsequently, the function was type checked using the new frontend implementation.

```
fun times_three (a: bits_t sz) : bits_t sz =>
  (a << 0b1) + a
```

Listing 15: Simple function from `collatz.v`

The Hoare infrastructure was then assessed through a proof, demonstrating functional correctness with respect to the multiplication of natural numbers. In detail, as depicted in Listing 16, the proof goal asserts that the returned value of this action is three times its input value. Notably, this input is specified by asserting that the local variable `a` holds the value `n`, which is feasible due to the fact that all parameters of a function are passed through the local context `Γ`, as mentioned in Section 4.4.2.

```
Theorem times_three_correct :
  ∀ sz n : nat,
  {{ Γ[a] = n }} times_three sz {{ r, r = 3*n }}
```

Listing 16: Correctness theorem for `times_three`

Using the BinOp rule as well as the Const and Var rules, the proof goal could be transformed into the following equality:

```
(bits(n) <<b 1) +b bits(n) = bits(3 * n)
```

Then, to formally establish this equality, additional lemmas were required to demonstrate the correspondence between operations on bit vectors and their effects on the encoded numerical value. For example, one such lemma proves that shifting a bit vector by one position is equivalent to multiplying its represented value by two.

Nevertheless, the necessity of additional lemmas, to verify the alignment of the implementation with its specification, is a separate concern. The Hoare logic, for its part, performed as intended, reducing the verification process to the proof of a single logical assertion.

5.2.2 Modularized Proofs

Building on this foundation, the next case study evaluates the modularization of proofs through a more complex example. In this regard, the queue implementation from Listing 12 is examined.

One of the verified theorems is illustrated in Listing 17. This theorem asserts that whenever a value n is enqueued, invoking the dequeue function will return the exact same value. For this statement to hold, it is assumed that the queue is initially empty, meaning the `empty` register is set to `1`. Furthermore, the value n is passed by asserting that a local variable a holds this value in the precondition and subsequently using this variable as a parameter to `enqueue₀`.

```
Theorem queue_correct :
  ∀ n: nat,
  {{ env[empty] = 1 ∧ Γ[a] = n }} <{ enqueue₀(a); dequeue₁() }> {{ r, r = n }}
```

Listing 17: Correctness theorem for queue implementation

Using the sequencing rule, this theorem can now be modularized using two lemmas. One reasoning over `enqueue₀` and another one over `dequeue₁`. These two statements can be seen in Listing 18 and Listing 19, respectively.

```
Lemma enq_correct :
  ∀ n: nat,
  {{ env[empty] = 1 ∧ Γ[a] = n }}
  <{ enqueue₀(a) }>
  {{ env[data] = n ∧ env[empty] = 0 }}
```

Listing 18: Correctness lemma for `enqueue0`

```
Lemma deq_correct :
  ∀ n: nat,
  {{ env[data] = n ∧ env[empty] = 0 }}
  <{ dequeue₁() }>
  {{ r, r = n }}
```

Listing 19: Correctness lemma for `dequeue1`

Assuming the validity of these lemmas, the proof of `queue_correct` follows very naturally from the application of the sequencing rule, as demonstrated in Listing 20.

```
Theorem queue_correct : ...
Proof.
  intros.
  eapply hoare_seq.
  - apply deq_correct.
  - apply enq_correct.
Qed.
```

Listing 20: Correctness theorem for queue implementation

Through the construction of this proof from smaller lemmas, it becomes easier to understand the proof structure and to identify the source of reasoning errors.

Notably, the Hoare infrastructure was even able to automate significant parts of the proofs for the two lemmas. In detail, it reduced them to the goals shown in Listing 21 and Listing 22, which were trivial to prove using the provided hypotheses.

```
env[empty] = 1 →
Γ[a] = n →
if env[empty]
  then Γ[a] = n ∧ 0 = 0
  else True
```

Listing 21: Proof goal of `enq_correct` after Hoare automation

```
env[data] = n →
env[empty] = 0 →
if neg env[empty]
  then env[data] = n
  else True
```

Listing 22: Proof goal of `deq_correct` after Hoare automation

5.2.3 Frame Problem

One issue that was encountered, during the evaluation of the proof infrastructure, was the well-known limitation of Hoare logic referred to as the *frame problem* [BMR93]. This problem describes the difficulty of utilizing first-order pre- and postconditions to explicitly specifying which parts of the system's state remain unchanged during an action's execution. Typical Hoare proofs reason only over specific portions of the overall state, leaving the effects on the remaining state unspecified. However, this lack of specification leads to challenges when attempting to combine seemingly unrelated proofs over actions which operate on disjoint parts of the environment.

To illustrate this challenge, consider the property involving two independent variable assignments, shown in Listing 23. Ideally, the proof of this property should be derivable from two separate proofs, each reasoning independently about a single assignment, as outlined in Listing 24.

```
Theorem assignent :
  ∀ n m: nat,
  {{ ⊤ }} <{ a := #n; b := #m }> {{ Γ[a] = n ∧ Γ[b] = m }}
```

Listing 23: Property over unrelated variable assignments demonstrating the frame problem

```
Lemma assign_a :                  Lemma assign_b :
  ∀ n: nat,                         ∀ m: nat,
  {{ ⊤ }}                           {{ ⊤ }}
  <{ a := #n }>                     <{ b := #m }>
  {{ Γ[a] = n }}                    {{ Γ[b] = m }}
```

Listing 24: Example of insufficient decomposition

Unfortunately, this decomposition is insufficient as the assignment lemma for `b` does not proof that the value of `a` remains unaffected by the execution. As a result, the sequence rule fails to guarantee the preservation of `a`'s postcondition, preventing the proof of the overall theorem.

An immediate solution to this problem, would be the adaptation of the lemma for `b`, to explicitly state that it does not interfere with the variable `a`. The adjusted version of this lemma can be seen in Listing 25.

```
Lemma assign_b :
  ∀ n m: nat,
  {{ Γ[a] = n }}
  <{ b := #m }>
  {{ Γ[a] = n ∧ Γ[b] = m }}
```

Listing 25: Example of insufficient decomposition

However, this solution primarily addresses the specific case rather than resolving the underlying issue. The problem may reoccur if the assignment of b is combined with a different action. The complexity of this issue arises from the fact that the complete set of variables is possibly unknown when formulating the lemma for b. Moreover, this approach reduces the readability of the statement by significantly inflating both, the precondition and postcondition.

6 Conclusion

This thesis investigated the scalability of formal verification methods for hardware design. A preliminary literature review revealed that conventional techniques, such as model checking and assertion-based verification, are insufficient for handling complex scenarios. As a result, this work focused on a theorem-proving-based approach. Specifically, it evaluated the capabilities of the Kôika HDL, identifying key limitations, including the lack of support for parametric designs and the absence of a modular proof infrastructure for complex verification tasks.

To address the first limitation, this work introduced a new parser for Kôika, leveraging Rocq's type checker to directly construct typed ASTs, as originally suggested by the creators of Kôika. However, this implementation went beyond the initial proposal by incorporating bidirectionality hints to resolve type-checking issues, ensuring the parser's practical usability.

The second limitation was addressed through the development of a new proof infrastructure based on Hoare logic. This framework enables modular reasoning over complex actions by decomposing proofs into smaller sub-proofs, each of which only focuses on a part of the original action, thereby enhancing scalability in hardware verification. Additionally, this infrastructure facilitates reasoning over parametric hardware designs, allowing for the construction of generalized proofs.

6.1 Limitations

While the proposed solutions present advancements in the formal verification of hardware designs, several limitations remain.

First, the new typed parser generates overly complex error messages when provided with an inconsistent hardware description. These verbose and sometimes unintuitive error messages make debugging more challenging, potentially hindering the development process and increasing the time required to identify and resolve issues.

Second, as mentioned in the evaluation, the implemented Hoare logic is subject to the frame problem, hindering the composition of certain proofs.

Furthermore, the current implementation of the proof infrastructure is limited to reasoning about individual Kôika actions. To enable verification on the specification level of complete circuits, additional lemmas are required to extend the framework's applicability to entire schedules. However, since these schedules combine multiple actions into a single design, this extension will also encounter the aforementioned frame problem.

Finally, while this work demonstrated the effectiveness of the proposed methodology on different case studies, its practical applicability to large-scale, real-world circuits remains an open challenge. The feasibility of the verification of highly parameterized architectures has not yet been fully explored.

Despite these limitations, the contributions of this thesis lay a solid foundation for hardware verification in Kôika.

6.2 Future Work

Future research could investigate various approaches to overcome the limitations identified in this work. One promising direction would be to extend the Hoare logic into a separation logic, which could address the frame problem by providing more control over state partitioning. Another avenue of exploration could involve evaluating the proposed framework on a larger hardware description, such as the RISC-V implementation from the original authors of Kôika. Additionally, to address a limitation of Kôika itself, future work could involve the specification of a temporal logic and the development of an associated proof infrastructure to enable robust formal reasoning over multiple execution cycles.

Bibliography

[Tsc+02] J. Tschanz *et al.*, "Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1396–1402, 2002, doi: 10.1109/JSSC.2002.803949.

[Fos20] H. Foster, "2020 Wilson Research Group functional verification study," 2020, [Online]. Available: https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study/

[19] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, vol. 0, no. , pp. 1–84, 2019, doi: 10.1109/IEEESTD.2019.8766229.

[BC13] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[Bou+20] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: a core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 243–257. doi: 10.1145/3385412.3385965.

[96] "IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language," *IEEE Std 1364-1995*, vol. 0, no. , pp. 1–688, 1996, doi: 10.1109/IEEESTD.1996.81542.

[88] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-1987*, vol. 0, no. , pp. 1–218, 1988, doi: 10.1109/IEEESTD.1988.122645.

[Mer+10] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010, pp. 179–188. doi: 10.1109/MEMCOD.2010.5558634.

[06] "IEEE Standard SystemC(R) Language Reference Manual," *IEEE Std 1666-2005*, vol. 0, no. , pp. 1–423, 2006, doi: 10.1109/IEEESTD.2006.99475.

[Gaj+00] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SPECC: Specification Language and Methodology*, 1st ed. Springer New York, NY, 2000, p. 313. doi: 10.1007/978-1-4615-4515-6.

[Var07] M. Y. Vardi, "Formal techniques for SystemC verification," in *Proceedings of the 44th Annual Design Automation Conference*, in DAC '07. San Diego, California: Association for Computing Machinery, 2007, pp. 188–192. doi: 10.1145/1278480.1278527.

[CNR13] A. Cimatti, I. Narasamdya, and M. Roveri, "Software Model Checking SystemC," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, pp. 774–787, 2013, [Online]. Available: https://api.semanticscholar.org/CorpusID:16897141

[GLD10] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 113–122, 2010, [Online]. Available: https://api.semanticscholar.org/CorpusID:16017959

[HPG15] P. Herber, M. Pockrandt, and S. Glesner, "STATE – A SystemC to Timed Automata Transformation Engine," *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1074–1077, 2015, [Online]. Available: https://api.semanticscholar.org/CorpusID:17324549

[Hoa78] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, doi: 10.1145/359576.359585.

[Dij75] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975, doi: 10.1145/360933.360975.

[SG88] J. Staunstrup and M. R. Greenstreet, "From high-level descriptions to vlsi circuits," *BIT Numerical Mathematics*, vol. 28, no. 3, pp. 620–638, Sep. 1988, doi: 10.1007/BF01941138.

[Dil96] D. L. Dill, "The Mur ϕ verification system," in *Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 390–393.

[Nik04] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, 2004, pp. 69–70. doi: 10.1109/MEMCOD.2004.1459818.

[Arm+19] A. Armstrong *et al.*, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019, doi: 10.1145/3290384.

[Pnu77] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. doi: 10.1109/SFCS.1977.32.

[FKL03] H. Foster, A. Krolnik, and D. Lacey, "Assertion Methodology," in *Assertion-Based Design*, Boston, MA: Springer US, 2003, pp. 21–56. doi: 10.1007/978-1-4419-9228-4_2.

[Das+06] S. Das, R. Mohanty, P. Dasgupta, and P. Chakrabarti, "Synthesis of System Verilog Assertions," in *Proceedings of the Design Automation & Test in Europe Conference*, 2006, pp. 1–6. doi: 10.1109/DATE.2006.243776.

[Arm+02] R. Armoni *et al.*, "The ForSpec Temporal Logic: A New Temporal Property-Specification Language," in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 296–311.

[05] "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language," *IEEE Std 1800-2005*, vol. 0, no. , pp. 1–648, 2005, doi: 10.1109/IEEESTD.2005.97972.

[CE81] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 52–71.

[QS82] J. P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351.

[BK08] C. Baier and J.-P. Katoen, "Principles of model checking," 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:5302889

[CH88] T. Coquand and G. P. Huet, "The Calculus of Constructions," *Inf. Comput.*, vol. 76, pp. 95–120, 1988, [Online]. Available: https://api.semanticscholar.org/CorpusID:15433820

[CH85] T. Coquand and G. Huet, "Constructions: A higher order proof system for mechanizing mathematics," in *EUROCAL '85*, B. Buchberger, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 151–184.

[Gor85] M. J. C. Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," 1985. [Online]. Available: https://api.semanticscholar.org/CorpusID:116922728

[HDL90] F. K. Hanna, N. Daeche, and M. Longley, "Veritas+: A specification language based on type theory," in *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, M. Leeser and G. Brown, Eds., New York, NY: Springer New York, 1990, pp. 358–379.

[CJ04] S. Coupet-Grimal and L. Jakubiec, "Certifying circuits in Type Theory," *Formal Aspects of Computing*, vol. 16, no. 4, pp. 352–373, Nov. 2004, doi: 10.1007/s00165-004-0048-3.

[Del00] D. Delahaye, "A Tactic Language for the System Coq," in *Logic Programming and Automated Reasoning*, 2000. [Online]. Available: https://api.semanticscholar.org/CorpusID:17394708

[BP21] T. Bourgeat and C. Pit-Claudel, "An experience report on writing usable DSLs in Coq," 2021.

[Bac+17] C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser, "Intrinsically-typed definitional interpreters for imperative languages," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017, doi: 10.1145/3158104.

[Ben+12] N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride, "Strongly Typed Term Representations in Coq," *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 141–159, Aug. 2012, doi: 10.1007/s10817-011-9219-0.

[Hoa69] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, doi: 10.1145/363235.363259.

[Hoa71] C. A. R. Hoare, “Procedures and parameters: An axiomatic approach,” in *Symposium on Semantics of Algorithmic Languages*, E. Engeler, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1971, pp. 102–116.

[Hoa72] C. A. R. Hoare, “Towards a Theory of Parallel Programming,” in *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, P. B. Hansen, Ed., New York, NY: Springer New York, 1972, pp. 231–244. doi: 10.1007/978-1-4757-3472-0_6.

[PM22] C. Pit-Claudel and F. Mathieu, “Kôika,” GitHub, 2022. [Online]. Available: https://github.com/mit-plv/koika/tree/c758c7b0092186f76ed858f4137366cc62f7a04a

[BMR93] A. Borgida, J. Mylopoulos, and R. Reiter, “On the Frame Problem in Procedure Specifications 1,” 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:263894208