A Type-Safe HDL Verified in Coq

Hanneli Carolina Andreazzi Tavante

School of Computer Science
McGill University
Montreal, QC, Canada

Submitted in partial fulfillment of the requirements for the
degree of Master of Science

Abstract

Hardware Description Languages (HDL), such as VHDL and Verilog, simplify the circuit specification, simulation, and synthesis by enabling different types of abstractions. Hardware verification pipelines reduce design faults caused by erroneous transformations of a design specification into the layout description. However, there is little work on the language aspect of Verilog itself, and designers tend to trust the language as a source of truth. Unfortunately, unverified languages may be unreliable and lead to circuit design faults. For instance, in Verilog, values can be converted automatically from one type to another when the context of use requires it, generating undesired bugs due to the automatic conversion.

In this thesis, we address the need for a verified, type-safe language that can rule out undesired faults in hardware projects occasioned by language issues. We present *Verifloq*, a strongly typed HDL based on a subset of the original Verilog language.

Verifloq is developed in the Coq proof assistant, and uses the Simply-Typed Lambda Calculus (STLC) as its core foundation. We develop a flexible small-step operational semantics for our language, and combined with its set of typing rules, we prove that *Verifloq* is a type-safe language.

We also provide several use cases for *Verifloq*, including a composed verification pipeline with Hoare Logic and a multi-staged hardware verification pipeline. Finally, we also present possible integration scenarios for High-Level Synthesis applications.

Résumé

Les langages de description matérielle (en anglais, Hardware Description languages, ou les HDL), tels que le VHDL et le Verilog, simplifient la spécification, la simulation et la synthèse des circuits en permettant différents types d'abstractions. Les pipelines de vérification matérielle réduisent les défauts de conception causés par des transformations erronées d'une spécification de conception en description de mise en page. Cependant, il y a peu de travail sur l'aspect langage du Verilog lui-même, et les concepteurs ont tendance à faire confiance au langage comme source de vérité. Malheureusement, les langages non vérifiés peuvent ne pas être fiables et conduire à des défauts de conception de circuit. Par exemple, dans le Verilog, les valeurs peuvent être converties automatiquement d'un type à un autre lorsque le contexte d'utilisation l'exige, générant des bogues indésirables dus à la conversion automatique.

Dans cette thèse, nous abordons le besoin d'un langage vérifié et de type sécurisé qui peut exclure les défauts indésirables dans les projets matériels occasionnés par des problèmes de langage. Nous présentons *Verifloq*, une HDL fortement typé basé sur un sous-ensemble du langage Verilog original.

Verifloq est développé dans l'assistant de preuve Coq et utilise le calcul lambda simplement typé (en anglais, Simply-Typed Lambda Calculus, ou le STLC) comme base de base. Nous développons une sémantique opérationnelle flexible à petits pas pour notre langage, et combinée avec son ensemble de règles de typage, nous prouvons que *Verifloq* est un langage de type sécurisé.

Nous fournissons également plusieurs cas d'utilisation pour *Verifloq*, y compris un pipeline de vérification composé avec le Hoare Logic et un pipeline de vérification matérielle en plusieurs étapes. Pour terminer, nous présentons également des scénarios d'intégration possibles pour les applications de synthèse de haut niveau.

Acknowledgements

People often insert happy stories and fond memories in this section. Sorry, this is not exactly what you will find here, but definitely there are some folks out there who made it all possible!

First, a sincere thanks to my advisor, prof. Zeljko Zilic. Prof. Zilic gave me the time and space to work on this project, even though it was an idea relatively far from his expertise. In every group meeting, he would give me room to talk about Coq, semantics, show eventually broken proofs, and comment on the papers I was reading. He showed me it is possible to do research regardless of your background, gender, race and previous work experiences.

Speaking of work environment, I'd like to thank my superb labmates! When Covid gave us less severe times, we started to meet in-person. Miguel, Katyayani, Perl and Guanyi gave me the moral support to write all the proofs for Verifloq while we went through some freshly baked brownies and papers on Tuesdays. You folks are the best. Also, huge thanks to Tom and the entire Kn0x team. Thanks for supporting our research group.

I don't know how I would have wrapped up my studies without the support of the therapists in the Wellness Hub at McGill; in particular, Devon Simpson. Thanks for taking care of my mental health!

During my failed attempts to join the PL community, I had the opportunity to meet a lot of people while supporting SIGPLAN conferences. I am very grateful to all the mentors I had, in particular to all my previous SIGPLAN-M mentors, who guided me during major academic crisis. A special shout out to C.C. [1] and their lab. C.C. adopted me as a long-term visitor to their group. In that very diverse environment, I managed to watch and participate in discussions about compilers, optimizations, static and dynamic analysis, etc. I also learned a lot about the inner mechanics of academia and received useful writing tips. These valuable suggestions helped me to become a better student.

Last, but not least, big thanks to V.R. and I.R. It is been more than 30 years, and they keep betting on me. Unconditional love. <3

[1] Not their initials in real life.

CONTENTS

List of Figures

1 Introduction

During the mid-1980s, the idea of Very-Large-Scale Integration (VLSI) became popular. As a direct consequence, robust Hardware Description Languages (HDL), such as VHDL and Verilog, were developed to describe the structure and behavior of digital circuits.

With the popularization of Very-Large-Scale Integration (VLSI) during the mid-1980s, robust Hardware Description Languages (HDL), such as VHDL and Verilog, were developed to describe the structure and behavior of digital circuits. HDL are similar to other popular programming languages, such as C and C++, but they heavily rely on the notions of time and concurrency, since they specify hardware. HDL aim to simplify the circuit specification, simulation and synthesis by enabling different types of abstractions: behavioral, register-transfer-level (RTL), and structural [Sagdeo 2007]. These abstractions help developers to increase their productivity and to reduce the number of bugs and faults in their designs.

Different types of faults can occur in hardware design [Kropf 1999]:

- *Design faults* are caused by an erroneous transformation of a design specification into the layout description. This type of issue can be mitigated with **hardware verification**;
- *Fabrication faults* are generated by layout defects during the fabrication process. Even if the design itself was correct, this type of fault might lead to unwanted behaviors in the circuit;
- *Faults during usage* are often an issue of old components or other types of physical damage that may occur after a period of usage.

[Kropf 1999] proposes two basic approaches to ensure that a given circuit does not contain design faults. The first one is the idea of *avoiding* faults via *correctness by construction* strategies. Specifically, a target design is established, it is necessary to generate or synthesize the correct circuit for the design. Hence, we also need guarantees about the synthesis environment, which should be equally verified. The second approach is to simply *detect* system faults, after a designated implementation on a certain abstraction level has been created. A reference description (a specification) allows us to verify the functional correctness of the circuit. We also need to state a correctness relation, which has to be established between the two descriptions.

Unfortunately, there is no silver bullet for tackling verification. A huge variety of functions, interfaces, protocols, and transformations must be verified, and is not possible to provide a general-purpose automated solution for verification. However, some parts of the verification process can be automated, especially when applied to a narrower application domain.

Among multiple verification strategies, [Joyce 1990] reports the well-grounded adoption of Higher-Order Logic (HOL) as a formalism for specifying and verifying hardware. HOL is the underlying formalism for popular theorem provers, such as HOL4 [2]. HOL supports parameterized hardware specifications by functions and by data types, and it can embed other types of logic, such as temporal logic. HOL is built on top of a small set of axioms, and it uses this axiomatization as a basis for guaranteeing proof security: every theorem generated by the HOL system is indeed a theorem of higher-order logic. We discuss HOL's versatility and wide adoption in hardware verification in Section 2.3. Nevertheless, HOL's simplicity comes with logical expressiveness limitations, and we have richer type theoretical approaches. For example, proof environments with constructive type theory and dependent

[2] https://hol-theorem-prover.org/

types are a promising take for hardware verification, as [Basin et al. 1991] and [Hanna et al. 1989] illustrate.

However, the fault classification proposed by [Kropf 1999] does not address potential issues with the HDL itself. Namely, designers tend to trust the language as a source of truth, but language construct design choices can lead to errors in the program, and subsequently in the circuit. Specifically, Verilog has a weak type system, and values can be converted automatically from one type to another when the context of use requires it. In certain situations, this type of automatic conversion leads to undesired bugs [Sudhakrishnan et al. 2008]. In short, this kind of fault is unrelated to circuit specification, but deeply connected with language aspects of the HDL. A prompt solution to this issue is the choice of an HDL with a strong type system, such as VHDL, where unwanted conversions do not happen.

Alternatively, High-level Synthesis (HLS) tools enable circuit description in a higher-level language (HLL), and then translate the code to a traditional HDL, such as Verilog or VHDL ([Coussy and Morawiec 2008]). For instance, Vivado HLS [3] offers C and C++ as HLL options, whereas Bluespec [Nikhil 2004] uses Haskell as a pure functional style for describing circuits. However, these implementations put emphasis on the HLL, but not on the aspects of the synthesized HDL.

1.1 Motivation

Motivated by the presence of faults generated by language design issues, this thesis implements a new HDL, *Verifloq*, based on a subset of Verilog. We develop a semantic model for the language that formalizes how the execution of each language construct should behave. Our starting point is the Simply-Typed Lambda Calculus (STLC), which we extend with

[3] https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html

other language constructs matching Verilog's syntax. *Verifloq* is strongly typed, as we provide designated types for every term. That means we can benefit from types to have a partial specification of the represented program.

Our language is *type-safe*, meaning that well-typed terms will always evaluate to a value or will follow a particular pre-established rule to be evaluated to another expression. In other words, well-typed terms will never reach a *stuck state*. Stuck states characterize a situation where the semantic model does not know what to do because the program has reached an unexpected or meaningless state [Pierce 2004]. Stuck states often represent a wide range of runtime errors in programming languages, such as segmentation faults and undefined behavior. We verify and prove the type safety property of *Verifloq* using the Coq proof assistant.

Finally, we chose Verilog as the reference HDL because it is still widely used by circuit designers as the primary HDL choice for hardware projects. Verilog is also a common synthesis target option in HLS tools, such as *Vivado HLS*.

Figure 1.1 shows the main components of *Verifloq*, summarizing the contents of Chapter 3. Our entire implementation is developed in Coq. *Verifloq* is compatible with a subset of the Verilog language. We accept plain Verilog files, and we provide a parser to *Verifloq*, which converts plain Verilog language constructs to *Verifloq*'s Coq terms. Alternatively, direct declaration of *Verifloq*'s Coq terms directly into the Coq is also accepted. We use the STLC as the baseline (Section 3.1) to develop an operational semantics (Section 3.2) and a set of typing rules (Section 3.3) for *Verifloq*. These two elements enable us to prove *Verifloq*'s type-safety property from progress and preservation core theorems (Section 3.4). Finally, with a verified type-safe language, we develop applications based on Hoare-style verification and discuss the usage in Program Equivalence for HLS (Chapter 4).

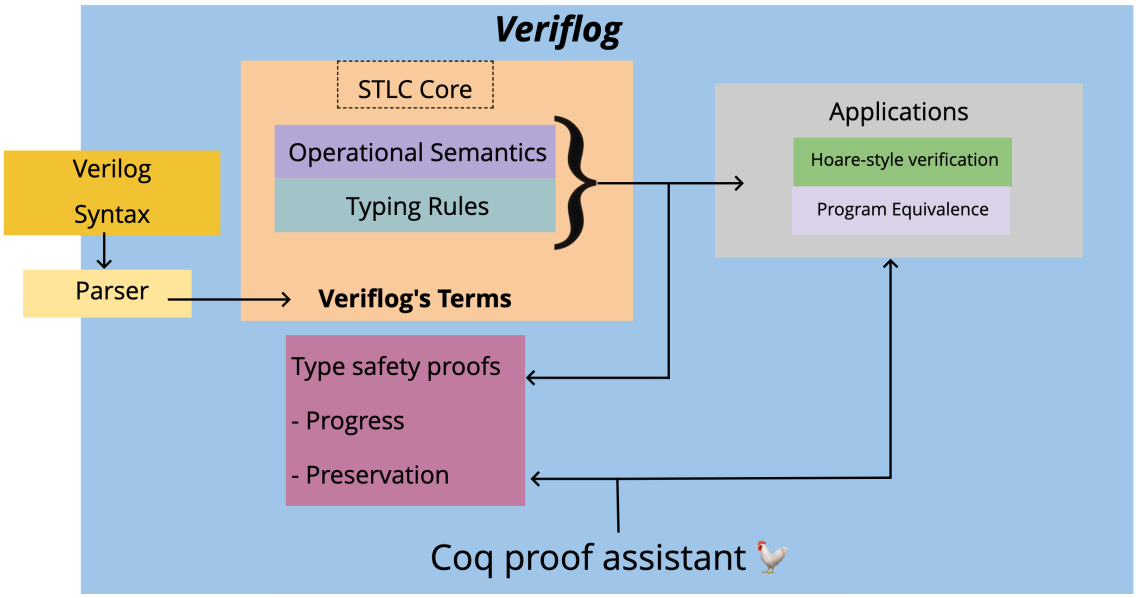

Fig. 1.1. Key components of *Verifloq*: we encode a subset of Verilog's syntax into Coq terms, and use the STLC to develop an operational semantics and typing rules to prove *Verifloq*'s type-safety.

1.2 Contributions

This thesis presents *Verifloq*, an HDL developed in the Coq proof assistant, which uses the Simply-Typed Lambda Calculus (STLC) as its core foundation. We develop a small-step operational semantics and prove our language is safe with respect to the proposed semantic model. With that, we rule out possible type mismatches or stuck terms in the language by providing a machine-checked, mathematical proof of type safety. *Verifloq* extends the hardware verification pipeline to prevent design faults by adding an extra layer of safety in the language level. This document demonstrates and discusses the following aspects:

(1) We introduce *Verifloq*, a new HDL directly supporting a subset of Verilog's language syntax.

(2) We let circuit designers write specifications as if they were writing pure Verilog, without the overhead of an entirely new language.

(3) We establish an operational semantics and a set of typing rules for *Verifloq*, making it a strongly typed language.

(4) We prove *Verifloq* is a type-safe language with respect to the proposed semantics using the Coq proof assistant.

(5) *Verifloq* is designed as an extensible formalism that can be combined with other types of verification mechanisms. For instance, we add an additional verification layer to *Verifloq* using Hoare Logic, enabling Hoare-style verification.

1.3 Thesis Outline

A short introduction to the Verilog language, its aspects, and common issues can be found on Chapter 2. We also extensively discuss different verification techniques for HDL. We then present multiple projects combining theorem provers and formal hardware verification. Finally, we set the stage for richer type theories and introduce the Coq proof assistant. In addition, we present the core calculus foundation of *Verifloq* - the Simply-Typed Lambda Calculus (STLC).

In Chapter 3, we present the development of *Verifloq* as a rich extension to the STLC. Then, we introduce *Verifloq*'s small-step operational semantics and the essential type safety proofs (progress and preservation).

Chapter 4 describes applications for *Verifloq*, including an additional verification layer with Hoare Logic and use cases for program equivalence of generated HDL from High-Level Synthesis (HLS) tools.

Chapter 5 discusses and compares different semantic approaches for HDL, and briefly presents the differences between *Verifloq* and other HLS formalization projects.

Finally, Chapter 6 reports the challenges in developing a project like *Verifloq*. It also lists multiple suggestions for future work.

2 Background

2.1 Hardware Description Languages and Verilog

Hardware Description Languages (HDL), such as Verilog, SystemVerilog, or VHDL, allow end-users to model digital systems (circuits) using different approaches. There are three well-known abstractions: behavioral, register-transfer-level (RTL), and structural [Sagdeo 2007]. The behavioral model holds the highest-level abstraction. It enables an algorithmic description of the circuit, allowing the user to describe synchronization between processes or blocks. RTL is an event-driven model that describes data transfer between registers. Lastly, the structural model expresses the hierarchy and inter-connectivity details of modules (netlists). These abstractions can be mixed within the scope of a project, and complement each other.

Verilog is a widely-adopted HDL that allows engineers to specify a digital system using high-level language abstractions. Proposed initially between 1983 and 1984 as a proprietary tool, Verilog has been standardized under IEEE 1364-2001 [IEEE 2001], and its reference manual contains extensive information about syntactic and semantic components of the language.

2.1.1 Describing circuits with Verilog

Circuit specifications in Verilog are declared in modules, which have input and output ports. Events in input ports cause output events. Modules can represent simple circuits, such as simple gate combinations, or complex systems. They can be specified behaviorally, via language constructs, or structurally, via hierarchical connection of submodules. Alternatively, modules' specifications can combine both abstractions.

In each module, it is possible to declare basic structures, such as continuous assignments (denoted by the keyword assign), simple data, expressions, and statements. Statements can be executed in sequential blocks. In general, standard module components are D-Type registers triggered by a clock edges (positive or negative). Listing 2.1 exemplifies the idea of basic modules, assignments and statements. The code contains a simple 4-bit counter in Verilog:

```
1  module counter4b (input clk, reset, output[3:0] counter);
2      reg [3:0] counter_up;
3      always @(posedge clk or posedge reset)
4      begin
5        if(reset)
6          counter_up <= 4'd0;
7        else
8          counter_up <= counter_up + 4'd1;
9      end
10     assign counter = counter_up;
11 endmodule
```

Listing 2.1. Simple counter in Verilog.

Verilog supports assignments with delays, meaning that changes on the inputs can be propagated to an output after a number of cycles. This particular type of assignment is called transport delay (or blocking transport delay). At most one change to a given wire can be scheduled at a time.

Non-blocking assignments describe delays behaviorally. They cause no delay in the current module. However, they schedule an assignment of a current value after a specified delay, enabling multiple changes to be scheduled to the same variable. A detailed

description of blocking and non-blocking assignments can be found on [Cummings 1999]. In Listing 2.1, we provided a code example of blocking and non-blocking assignments. Blocking statements are denoted by =, on line 10, and non-blocking statements are denoted by <=, as lines 6 and 8 show.

Verilog supports basic datatypes such as integers, reals, strings, and vectors. Vectors of vectors are called memories. Verilog also supports default imperative language constructs, such as if/else blocks and while loops.

To connect all the previously described concepts about Verilog and HDL, Listing 2.2 shows the implementation of a single port RAM, where we can read and write data.

```
1  module single_port_RAM (parameter addr_width = 2, data_width = 3)
2      ( input wire clk,
3        input wire [addr_width-1:0] addr,
4        input wire [data_width-1:0] write,
5        input wire write_enable,
6        output wire [data_width-1:0] read );
7    reg [data_width-1:0] ram[2**addr_width-1:0];
8    always @(posedge clk)
9    begin
10     if (write_enable == 1)
11       ram[addr] <= write;
12   end
13   assign read = ram[addr];
14 endmodule
```

Listing 2.2. Single port RAM in Verilog.

In general, hardware implementations consist of different design units that always run concurrently, and HDL must model this behavior correctly. The states of some units may imply waits, but they will provide certain outputs as a function of inputs and current state, independent of other units. For instance, gates, module instances, and RTL assignments are continuously providing the outputs as a function of inputs. This design paradigm is different from other languages, such as C or Java.

A more detailed guide to the Verilog language can be found on [Sagdeo 2007], and [Taraate 2022] provides a deeper discussion on the concurrent characteristics Verilog operators.

Finally, Verilog is a weakly-typed language. In a weak type system, the type of an expression carries little or no information about the denoted object. Values can be converted automatically from one type to another when the context of use requires it, and there are situations where the automatic conversion leads to surprising or unwanted results. This behavior is present in other popular languages, such as JavaScript [Thiemann 2005]. VHDL, on the other hand, has a so-called strong type system. Namely, it does not allow operations on data that are of incompatible types.

2.2 HDL and Verification

In the context of HDLs, **verification** is a process to demonstrate that a particular design and a set of specifications are preserved during implementation [Bergeron 2000]. It ensures the results of one or more transformations correspond to what designers expected. Transformations can be any set of processes that take one or more inputs and produce outputs. Analyzing the inputs with the expected results and matching them against the actual outputs

in an iterative process is called **reconvergence model**. This strategy is essential for ensuring product quality, preventing bugs, and setting standards in commercial or prototyping scenarios. It can also save time and resources by eliminating recalls or forced updates.

[Kropf 1999] makes an important distinction between **verification** and **validation**. Validation is the process of gaining confidence in the specification by examining the implementation's behavior, and it generally does not involve reasoning about transformations or specifications. Validation is often achieved by simulation.

There are different paths for verifying digital circuits depending on what designers aim to verify, and we summarize these distinct aspects in the upcoming subsections.

2.2.1 Formal and Functional verification

Functional verification ensures a particular design implements an intended functionality [Bergeron 2000]. It is often done via observed results. A popular technique is "black-box" verification, where implementation details are not shared, and the verification is accomplished through available interfaces. Although it lacks visibility, it is very convenient for being implementation-agnostic, and this technique can be used in multiple contexts. It is also helpful for decoupling results from implementation specifics.

On the other side of the spectrum, the "white-box" verification model enables total visibility of the implementation details, allowing the users to quickly combine and try different states and inputs. However, its immediate downside is the lack of generalization, making it tightly coupled to one particular design.

Different "box" models are not mutually exclusive, and can complement each other in valuable aspects. It is fairly common to find both "black-box" and "white-box" models as different verification steps for the same project. [Howar et al. 2019] reports a recent example of this style combination for software and hardware domains. Regardless of the

chosen technique, functional verification can only show that a design meets the intent of its specification by plain evidence. Nevertheless, it cannot mathematically prove anything about the process or the obtained results.

Circuit design correctness can also be verified through well-founded ideas from *formal verification* [Kropf 1999]. Specifically, there are two principal approaches. *Equivalence checking* validates equality of states and/or steps on all design transformations involved in the process [Bening and Foster 2001]. For example, it can ensure that a reference RTL model is logically equivalent to a transformed or refined model (see [Koelbl et al. 2009] and [Foster 2001]). Equivalence checking can also be applied to comparing output netlists [Kam and Subrahmanyam 1995] and matching netlists against an expected RTL code [Vasudevan et al. 2006].

Another common technique in formal verification is *Model Checking* [Clarke et al. 2009]. It enables the systematic checking of one or multiple properties under a finite-state, logical model for circuits. The foundations of model checking are derived from [Hoare 1969]. This foundation work uses deductive reasoning to explore the mathematical logic underlying computer programming. By specifying an initial set of axioms and a set of inference rules derived from these axioms, all results from running programs can be proven correct. The meaning of a statement is defined in terms of its effects, and they are materialized via assertions that can be made about the associated program or design. Assertions are widely adopted in hardware verification, and they have their own particularities and categorizations in the verification domain.

2.2.2 Dynamic and Static verification

The idea of hardware verification through assertions is the foundation for the technique called **Assertion Based Verification** (ABV). In this model, assertions are high-level statements formulated under a certain type of logic (temporal, linear, or linear temporal logic, for example), and they encode how the circuit, or parts of it, should behave [Foster et al. 2004]. Assertions are highly versatile and can be added in multiple stages of the design process. Designers can include new assertions incrementally, according to the needs of the project. Such flexibility brings unit-wise and system-wise benefits. Faulty or buggy implementations can be detected locally or during integration stages [Tao 2009]. Hardware assertions are commonly written in some kind of specification language; typical examples are the Property Specification Language (PSL) [IEEE 2005a] and SystemVerilog Assertions (SVA) [IEEE 2005b]. These languages take declarations of linear temporal relations between signals and rely on a checker generator to emulate the expected behavior. A checker generator synthesizes monitor circuits from assertions, and an assertion checker, expressed in HDL, captures the behavior of a given assertion [Boulé and Zilic 2008]. The assertion checker produces assertion signals with the results of executed assertions. Assertion checkers are also known as Assertion Circuits. Figure 2.1 summarizes the main components and steps of ABV.

Assertions can be used **dynamically** to offer a solid support for random testing [Bird and Munoz 1983]. Manually and automatically generated test-benches [Biederman 1997], for example, compose simulation code that creates a predetermined input sequence to be matched against an expected output. Simulation of all possible outcomes is a common **dynamic** verification technique that behaves well for small circuits.

However, purely dynamic techniques do not scale up for larger projects, given the excessive amounts of possible combinations [Eastham and Thirunarayan 1996]. In contrast, ABV can be used in a **static** verification scope. In this static scenario, assertions are

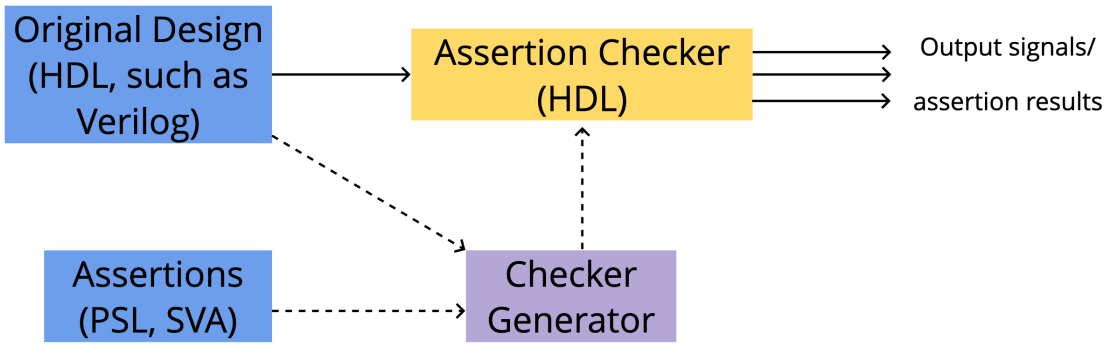

Fig. 2.1. Main components of ABV: original circuit, assertions, checker generator, assertion checker and results.

combined with formal methods techniques. For instance, model checking and assertions have a long history of applications. As an example, [Raik et al. 2008] reports the use case of model checking to detect untestable faults in RTL models. The project can automatically generate a corresponding assertion in PSL from the outputs to reveal issues with the designated sequential synchronous design.

Another example of static verification consists of processing assertions with the support of theorem provers. More robust than pure automaton checkers, they may rely on more expressive types of logic, which can be seen as a powerful semantic framework. In contrast with the finite state machine (FSM) approach, which usually targets only RTL models, some theorem provers are flexible enough to encode all the circuit formalisms - behavioral, RTL, and structural. To illustrate from a historical perspective, works from the early 2000's model a subset of the PSL language using the Prototype Verification System Prover (PVS) [Shankar et al. 2001]. In the work of [Morin-Allory and Borrione 2006], authors synthesize provably correct monitors (observed input value sequence signals) that match a designated temporal expression. In [Morin-Allory et al. 2008], authors take a different verification

path. Their project encodes PSL's syntax and semantics in PVS, and with that, they prove the correctness of a small set of rewrite rules.

2.3 Theorem Provers and Hardware Verification

The previous examples and use cases are directed to PVS, but other theorem provers have been largely adopted to support hardware verification. As a concrete example, in the 1990s and early 2000s, Isabelle [Paulson 1994] and the entire HOL [Gordon 1988] family received significant attention from the hardware verification community. [Kropf 1999] dedicates an entire chapter of their work to compare verification techniques in HOL versus plain FSM. This reference provides a well-structured introduction to proofs using predicate and first-order logic (FOL), as well as axiomatic theory principles applied to hardware verification based on [Hoare 1969]. In a fruitful discussion about the limitations of FOL, [Kropf 1999] demonstrates the impossibility of binding functions by quantifiers and the separation between predicates and terms in this type of logic. These arguments justify the need for Higher-Order Logic.

Language-wise, Higher-Order Logic is a rich construction carrying the notions of types (tags for non-empty sets, in this context), functions, lambda-abstractions (notion derived from the λ-calculus, see [Engeler 1984]), polymorphism, bound and free variables. [Kropf 1999] systematically drafts a compact semantics for Higher-Order Logics targeting hardware verification, which is composed of constants, function application and conditionals. Authors also show how to increment this rudimentary system with extensions that preserve the correctness of the model (known as conservative extensions). Given the required axioms and semantics, it becomes possible to state theorems (and not simple assertions) which carry the goals to be proved. Initially described on [Kumar et al. 1991], a generalization for the hardware verification procedure using Higher-Order Logic is stated as follows:

(1) Create a formal specification;

(2) Create a formal implementation;

(3) State the proof goals and theorems;

(4) Perform the correctness proof.

Distinct theorem provers may require different strategies for hardware verification. As additional examples, [Aagaard et al. 1993] develops Nuprl [Constable et al. 1986] tactics for automating frequently required verification tasks. Nuprl's type system is based on a sequent version of Martin-Loof's constructive type theory, which is more expressive than HOL's simple type system. A detailed survey on formal verification strategies with multiple theorem provers from the early 2000's, not restricted to HOL and PVS, can be found on [Kern and Greenstreet 1999].

Since mid-2000s, several research projects on hardware verification began targeting High-Level Synthesis (HLS). In HLS, the circuit specification is stated in higher-level language (HLL), such as C or C++, and then it is transcompiled into RTL design, often using an HDL such as Verilog or VHDL. Chapter 5 reports a broader discussion on HLS and how the techniques relate to the work in this thesis.

In the last 20 years, other theorem provers were developed or gained space in the formal methods community. Hence, they also became an option for hardware verification. For instance, in [Guo et al. 2016], authors combine a model checker with the Coq proof assistant [4] ([Barras et al. 1997], [Coquand and Huet 1985]) for verifying system-level security on System-on-Chip (SoC) designs. In [Jin et al. 2017] and [Bidmeshki et al. 2017], authors develop a Verilog-to-Coq library that generates security property theorems for circuits. Their project, *Vericoq*, aims to facilitate the development of hierarchical proofs. It enables

[4] https://coq.inria.fr/

the verification of HDL code and the corresponding reusable lemmas for each module. Vericoq uses its own axiomatic semantics to generate the translation between Verilog and Coq.

Verifloq, the main project in this thesis, also uses the Coq proof assistant for developing a formalization for a subset of the Verilog language. In Section 2.4, we provide an overview of Coq.

2.4 Brief Introduction to the Coq Proof Assistant

Coq is a proof assistant for higher-order logic, along the lines of HOL (see Section 2.3). It allows the verification of programs against a formal specification via mathematical proofs. Coq's foundational theory/logic is based on the Calculus of Inductive Constructions (CIC) [Coquand and Huet 1988]; more specifically in an extension named *Gallina*. In a high-level overview, it combines predicate calculus, inductive predicate definitions, and recursive function definitions. Coq enables program extraction from the constructive contents of proofs. In this sense, Coq is a robust tool that takes advantage of proof objects and gives them a computational interpretation.

Coq is also a dependently typed language. That means it can include references to programs inside of types. The classic example of dependent types is the type of arrays, where the array type includes information about its dimensions. The idea of dependent types is not new (see [Pierce 2004]), and it can express extremely precise specifications of program behavior. "Dependent types also often let you write certified programs without writing anything that looks like a proof" [Chlipala 2013].

Formalizations in Coq may benefit from multiple language safety guarantees for free. For example, *Gallina* ensures that a computation always terminates (a property known as strong normalization) [Bertot and Castéran 2004] - hence, Coq code properly developed also

inherits this property. Coq is also heavily driven by **tactics** for interactive proof development. Tactics are commands that decompose proofs goals into simple ones or sometimes solve a goal entirely. They are a convenient way to handle complex proofs. Tactics are also present in Nuprl, HOL, and Isabelle. A summary of useful Coq tactics can be found on Appendix B.4.

2.5 Programming Languages and Semantic Models

This thesis presents the development of type-safe language, *Verifloq*. The implementation process requires specifying the syntactic constructs (syntax), as well as how the language behaves, and what each of its constructs means (semantics).

By establishing a formal semantics to a language, we aim to develop its mathematical model to understand and reason about how programs written in it behave. "Trying to define the meaning of program constructions precisely can reveal all kinds of subtleties of which it is important to be aware" [Winskel 1993]. A semantic model is also the basis for analysis and verification.

Semantic models for languages are often categorized in three groups ([Schmidt 1996], [Pierce 2002]):

(1) **Axiomatic semantics**: The meaning of a well-formed program is a logical proposition (or specification) that states some property about the input and output. In other words, this model fixes the meaning of a programming construct by giving proof rules to it, within a particular type of logic. Instead of first defining the behaviors of programs and then deriving laws from this definition, axiomatic methods take the specifications as the definition of the language. The meaning of a term is just what can be proved about it. This is the semantic style we see in Hoare Logic. The Hoare-style verification

system we developed in Section 4.1 is an example of axiomatic semantics, and the proof rules we defined for each construct combined with the pre and post conditions exemplify how this system works. Another example of an axiomatic approach is *Vericoq* (see [Jin et al. 2017] and [Bidmeshki et al. 2017]), previously mentioned in Section 2.3.

(2) **Operational semantics**: The meaning of a well-formed program is the trace of computation steps that results from processing the inputs. In other words, the behavior of a programming language is defined by an abstract machine for it. A state of the machine is usually a term, and a transition function defines the behavior. For each state, this transition function either gives the next state (by performing a step of simplification on the term) or declares that the machine has halted. There are different styles of operational semantics, and it is not uncommon to have multiple ones for a single language. For example, in *Structural Operational Semantics (SOS)*, [Plotkin 2004], as a computation proceeds, "branches of the abstract syntax tree are gradually replaced by the values that they have computed. In an initial state the program tree is purely syntactic, in a final state it has been replaced by its computed value, and in between, it is usually a mixture of syntax and computed values" [Mosses 2001]. This operational semantic style is also known as *Small-Step Operational Semantics*.

Natural semantics or *Big-Step Operational semantics*, introduced by [Kahn 1987], is a special case of the small-step semantics, involving initial and final states, but no intermediate states. This style may be easier to get started, as it hides intermediate state information. However, it generally does not allow any sort of reasoning about concurrent behavior. Other operational semantic approaches are *Modular Operational Semantics* [Mosses 1999] and *Reduction Semantics* [Felleisen and Friedman 1987].

(3) **Denotational semantics**: The meaning of a well-formed program is a mathematical function from input data to output data. It emphasizes that a program has an underlying mathematical meaning that is structured on the language's syntax definition [Winskel 1993]. Denotations are typically higher-order functions between complete partial orders. "The semantics of a complete program is its observable behavior, which is obtained from its denotation." [Mosses 2001]. The original denotational style was developed by [Scott 1972], and typically, denotations are functions of environments and continuations. *Monadic Semantics* [Moggi 1991] is another denotational style, based on category theory.

Elaborating a semantic model is a crucial step in language formalization. Instead of elaborating an entorely new model, we often decide to build on top of other existing computational formalisms. Section 2.6 describes the Simply-Typed Lambda Calculus, the foundation of strong types for *Verifloq*.

2.6 The Simply Typed Lambda Calculus

The pure (untyped) *Lambda Calculus* (or λ*-calculus*) was proposed by Church ([Church 1932]) as a general theory of functions and logic intended for the foundations of mathematics. All functions computable by Turing machines can be represented in the Lambda calculus. The untyped version of this calculus, also known as "type-free theory", allows every expression (a function) to be applied to every other expression (any argument) [Barendregt 1993]. In the untyped Lambda calculus, all language constructs are functions, and there are no side-effects or state. The language is built on top of *lambda terms*, which are syntactically valid expressions. The simplest version of the untyped Lambda calculus is built

with three base terms: a set of variables V, function application and function abstraction λ, which are presented in Definition 2.1.

Definition 2.1 (Base terms of the untyped Lambda Calculus.).

t ::= (Terms)

| V (Set of Variables)

| λ x $\cdot$ t, $x \in$ V (Function abstraction)

| t t (Function application)

Variables, such as x, y, z, are lambda terms. If F and G are lambda terms, their *function application*, FG, is also a lambda term. If x is a variable in the set of variables V and t is a term, the *function abstraction* $\lambda x \cdot t$ is also a lambda term. x is the function parameter, and t is the body of the function.

The *substitution* operation handles the computations in the Lambda Calculus. It is responsible for the replacement of a variable y in a term r by another term s. There are two important rules related to substitution: α*-reduction*, which simply renames bound variables in a term, and β*-reduction*, a recursive procedure that replaces all free occurrences of variable x in a term E by another term F. Bound variables in E are renamed by α-reduction (if necessary) to avoid capturing free variables in F.

In the untyped Lambda calculus, there is no restriction on how we can use functions. This loose behavior can lead to bad combinations of terms. For example, what would it be the *successor* of a boolean? The function *successor* would make sense for natural numbers, but not for $true$ or $false$.

To address this issue, typed versions of the Lambda calculus ([Curry 1934], [Church 1940], [Barendregt et al. 2013]) were proposed, and among them, we find the Simply Typed Lambda Calculus (STLC). Types are objects of a syntactic nature and may be assigned to

lambda terms. Types provide a partial specification of the represented algorithms and are useful for demonstrating partial correctness. A typed lambda-calculus has a language in two parts: the language of types (typing rules), and the language of terms (a set of rules to manipulate these terms) [Loader 1998]. Terms are equally expressive as HOL terms.

The simplest form of the STLC has only the type of functions as its single type [Barendregt et al. 2013], as presented in Definition 2.2.

Definition 2.2 (Types in pure STLC.). $\tau := \mathrm{T} \to \mathrm{T}$

In the STLC, we revisit the base terms from the untyped Lambda Calculus, and assign them corresponding types [Barendregt et al. 2013]. Namely, function abstractions will have a designated type. To assign a type to a term, we use the notation $x : T$, meaning that x has type T, as stated in Definition 2.3.

Definition 2.3 (Terms in pure STLC.).

t ::= (Terms)

| V (Set of Variables)

| λ x:T $\cdot$ t, $x \in$ V (Function abstraction)

| t t (Function application)

The STLC relies on a function mapping a finite set of variables into the set of types. This is named *context*, and it's often represented by the Greek letter Γ. Γ represents a typing declaration in $x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n$, meaning it maps each variable x_n to a corresponding type T_n.

Let t be any term in the STLC, and T be any type. We define the ternary relation of typing rule $\Gamma \vdash t : T$ for each term in the language as specified in Definition 2.4. These rules restrict what expressions are evaluated. Given a typing context Γ and expression e, if there

is some type T such that the previously stated ternary relation $\Gamma \vdash t : T$ holds, we say that e is *well-typed* under the context Γ. If Γ is the empty context, we say e is well-typed.

Definition 2.4 (Typing rules in pure STLC.).

$$\frac{x : T \in \Gamma}{\Gamma \vdash x \in T} \text{ TRVar}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x\ T_1 \cdot t : T_1 \to T_2} \text{ TRAbs}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1} \text{ TRApp}$$

Lastly, to formalize the execution of a language, in this case the STLC, we must select a semantic style among those described in Section 2.5. We can then verify properties such as *type safety*: well-typed terms can never reach a stuck state during evaluation.

In Chapter 3, we will connect the concepts seen in this section by developing a type-safe language, *Verifloq*, on top of the STLC, verified in Coq.

3 Verifloq: A Coq formalization for Verilog

In this section, we put together the base concepts seen in Chapter 2 by presenting the development of a new HDL. We formalize a small set of the Verilog language using the Simply Typed Lambda Calculus (STLC) as the core foundation for our verification strategy. **Verifloq**'s implementation follows the approach described on [Pierce et al. 2021].

Verifloq supports a subset of the original Verilog language, and the implemented constructs are defined in Figure 3.1.

⟨*v*⟩ ::= bool | Nat

⟨*op*⟩ ::= + | *
  | <
  | >... (Mostly basic arithmetic operations)

⟨*expr*⟩ ::= ⟨*v*⟩
  | x (variables)
  | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩

⟨*stmt*⟩ ::= ⟨*stmt*⟩ ; ⟨*stmt*⟩
  | if ⟨*expr*⟩ then ⟨*stmt*⟩ else ⟨*stmt*⟩
  | ⟨*expr*⟩ = ⟨*expr*⟩
  | ⟨*expr*⟩ <= ⟨*expr*⟩

⟨*mdl*⟩ ::= [always_ff @ (posedge clock) ⟨*stmt*⟩]

Fig. 3.1. Verifloq's syntax.

In Section 3.1, we describe the implementation of Verifloq by expanding the STLC to support the corresponding Verilog's constructs. The development and details of *Verifloq*'s small-step operational semantics are defined in Section 3.2. The typing relations are stated in

Section 3.3. Those are the necessary development steps to prove that *Verifloq* is a type-safe language. Specifically, in Section 3.4, we show that well-typed terms can never reach a stuck state during evaluation. Lastly, Section 3.5 brings some code examples, and Section 3.6 provides a quick summary of the development process.

3.1 Verifloq: Incrementing the STLC with Verilog Components

In *Verifloq*, we extend the pure STLC with Verilog's language constructs. Besides the original Arrow type, we add other simple **base types** (sets of simple, unstructured values). *Verifloq* currently supports natural numbers (Nat), booleans (Bool) and the unit type (Unit). The Unit type carries the notion of mutable state [Wadler 1992], and this is the type that typically represents side effects in a language. Side effects can modify the global program state. I/O operations, such as reading and writing from the disk, are examples of functions with side-effects. We implement Unit for future *Verifloq* extensions and at this time, side effects are not a major component of *Verifloq*'s formalization.

We also add multiple **derived types**, a composition of types that can be obtained from the combination of other types. Specifically, we add support for generic typed lists (List T), sums (Sum), tuples (Tuple), records (Record) and assignments (Assignment).

Sum types (denoted by T_1+T_2) represent heterogeneous collections of values, such as a node in a binary tree, which can be either a leaf or another inner node with other children. Tuples or Pairs, also known as product types $T_1 * T_2$, take the form of (x:T1, y:T2). (13, true) is an example of a 2-ary tuples of type (Nat, Bool). Records are a generalization of tuples. In our context, they are useful for storing signal declarations. Records are a collection of labelled fields, represented as $\{l_1 : T_1, ..., l_n : T_n\}$. In our implementation, they carry blocking and non-blocking assignments. In *Verifloq*, modules correspond to Variant types in the literature. These are a generalization of Sum types, or, in other words, a generalization of

n-ary labelled types. In our context, they will carrey the type of evaluated records throughout time Variant types are represented as $\langle l_1 : T_1, ..., l_n : T_n \rangle$ in our semantic model, and $l_1, ..., l_n$ are field labels.

The type constructs of *Verifloq* are stated in Listing 3.1, and the corresponding type definitions in Coq code can be found in Listing 3.2.

τ := (Types)

| T $\rightarrow$ T (Function Application)

| Nat

| Bool

| Unit

| Sum (T_1+T_2)

| Pair (T_1, T_2)

| List T (Generic List)

| Record ($T_1, T_2, ..., T_n$)

| ModuleAssignment ({ $l_1 : T_1, ..., l_n : T_n$ })

Listing 3.1. Types in Verifloq.

```
Inductive typ : Type :=
  (* Pure STLC *)
  | Ty_Arrow : typ -> typ -> typ
  (* Base types *)
  | T_Nat : typ
  | T_Bool : typ
  | T_Unit : typ
  (* Derived types *)
  | T_Sum : typ -> typ -> typ
  | T_List : typ -> typ
  | T_Tupl : typ -> typ -> typ
  | T_Record : typ -> typ -> typ
  | T_ModuleAssignment : typ -> typ -> typ.
```

Listing 3.2. Verifloq's types in Coq.

The next step consists of expanding the terms accordingly. Besides the three base terms for the pure STLC, we add terms for the natural numbers and their respective unary and binary operations, general recursion, blocking, and non-blocking operations, as well as modules. Listing 3.3 shows the inductive definitions of all supported terms in our language.

In *Verifloq*, we only allow synchronous, positive edge modules. Currently, only one module is supported. Register bits should be specified all at once. Vector resizing is also not supported.

```
Inductive tm : Type :=
  (* pure STLC *)
  | tm_var : string -> tm
```

```
| tm_app : tm -> tm -> tm
| tm_abs : string -> typ -> tm -> tm
(* binary and arithmetic operations *)
| tm_input : string -> tm
| tm_output : string -> tm
| tm_const: nat -> tm
| tm_succ : tm -> tm
| tm_pred : tm -> tm
| tm_add : tm -> tm -> tm
| tm_sub : tm -> tm -> tm
| tm_lt : tm -> tm -> tm
| tm_gt : tm -> tm -> tm
| tm_lte : tm -> tm -> tm
| tm_gte : tm -> tm -> tm
| tm_eq : tm -> tm -> tm
| tm_or : tm -> tm -> tm
| tm_and : tm -> tm -> tm
| tm_mult : tm -> tm -> tm
| tm_div : tm -> tm -> tm
| tm_neg : tm -> tm
| tm_not : tm -> tm
(* if *)
| tm_if : tm -> tm -> tm -> tm
(* unit *)
| tm_unit : tm
(* booleans *)
```

```
| tm_true : tm
| tm_false : tm
(* sums *)
| tm_inl : typ -> tm -> tm
| tm_inr : typ -> tm -> tm
| tm_case : tm -> string -> tm -> string -> tm -> tm
(* lists *)
| tm_nil : typ -> tm
| tm_cons : tm -> tm -> tm
| tm_lcase : tm -> tm -> string -> string -> tm -> tm
(* tuples *)
| tm_pair : tm -> tm -> tm
| tm_fst : tm -> tm
| tm_snd : tm -> tm
(* recursion *)
| tm_rec : tm -> tm
(* records *)
| tm_emprec : typ -> tm
| tm_record : tm -> tm -> tm
| tm_lproj : tm -> tm -> string -> string -> tm -> tm
(* assignments *)
| tm_empasgn : typ -> tm
| tm_asgn : typ -> tm -> tm -> tm
| tm_casgn : tm -> tm -> string -> tm -> string -> tm -> tm
| tm_blocking : tm -> tm -> tm
| tm_nonblocking : tm -> tm -> tm
```

```
| tm_module : tm -> tm -> tm -> tm -> tm.
```

Listing 3.3. Terms in Verifloq.

We must convert the input Verilog syntax to their respective Coq terms. Notation conversion is achieved with Coq's Notation command. Notation works like a macro, converting a designated syntax to a respective inductive definition in Coq. Listing B.1 shows how we convert the traditional if/else syntax from Verilog to Coq encoding using Notation as an example. It is necessary to specify one notation per language construct.

3.1.1 Verilog Programs Converted into Verifloq Terms

Listing 3.4 illustrates a simple Verilog program, and Listing 3.5 shows its respective translation to *Verifloq*'s terms, using the definitions presented at Listing 3.3.

```
module HalfAdd(a,b,sum,carry);
  input a,b;
  output sum,carry;
  xor(sum,a,b);
  and(carry,a,b);
endmodule
```

Listing 3.4. Half-adder example in Verilog.

```
( tm_module (tm_var "HalfAdd") (tm_const 0)
  (tm_record
    (tm_tuple [ (tm_input "a"), (tm_input "b") ],
      [ (tm_output "sum"), (tm_output "carry") ]
    )
    sum :::= (tm_xor a b)
    carry :::= (tm_and a b)
  )
  []
)
```

Listing 3.5. Half-adder example translated to *Verifloq*'s terms.

Note the module contents in Listing 3.5 are defined within a record type. tm_const 0 indicates the clock for the module, which will be always a positive, synchronous edge. The first element of the module is a tuple of input and output wires. Since we have multiple inputs and outputs, the tuple carries a list of tm_input and tm_output terms. The syntactic element :::= represents variable assignments. Lists, represented by tm_nil and tm_cons, have the standard representation of [].

In Section 3.2, we present the formalization to execute the computations in a *Verfilog* program, using our small-step operational semantics.

3.2 Verifloq's Small-step Operational Semantics

In this thesis, we propose a small-step operational semantics for *Verifloq*. Reiterating the definitions in Section 2.5, an operational semantics consists of an abstract machine for the language, and we define rules to describe how each term, initially in a pure syntactic form,

inductively steps into the program evaluation until it reaches a computed value. To specify the small-step operational semantics for *Verifloq*, we execute the following items:

(1) Define a set of values for the language;

(2) Define the notion of free-variables in the system;

(3) State the substitution relation;

(4) State the small-step relation (reduction).

Values are elements that cannot be reduced any further, such as: true, false, and natural numbers. *Verifloq*'s base values are defined in Listing 3.6. When dealing with function abstractions (i.e. $\lambda : T \cdot t$), we have a choice: either always treat them as values, or only consider them as values after t, the body of the function, has been reduced to a value. We take the traditional approach of the STLC and use the first option; namely, we always treat lambda abstractions as values. This choice is also convenient for defining our step relation, and we choose to work with only closed terms (terms with no free variables). This maneuver simplifies the substitution operation, to be stated shortly after.

```
Inductive val : tm -> Prop :=
  (* Pure STLC *)
  | v_abs : forall x T t1,
      val (tm_abs x T t1)
  (* Extensions *)
  | v_nat : forall n : nat,
      val (tm_const n)
  | v_inl : forall v T1,
      val v ->
      val tm_inl T1 v
  | v_inr : forall v T1,
```

```
    val v ->
    val tm_inr T1 v
| v_lnil : forall T1, val tm_nil T1
| v_lcons : forall v1 v2,
    val v1 ->
    val v2 ->
    val tm_cons v1 v2
| v_unit : val tm_unit
| v_pair : forall v1 v2,
    val v1 ->
    val v2 ->
    val tm_pair v1 v2
| v_true : val tm_true
| v_false : val tm_false.
```

Listing 3.6. Base values of our implementation.

We now state the *substitution* operation, which replaces a given term y for occurrences of some variable x in another term t, such that $[x := y]\ t$. It reads as "substitute y for x in the term t". The substitution operation needs to be stated for all existing terms, and the implementation of this recursive function can be found in Listing B.2.

In Figure 3.2 and Figure 3.3, we state the step relation for function abstractions, function application, recursion, natural numbers and boolean terms. Some trivial binary operations, such as multiplication or comparison of natural numbers, are omitted for brevity. They will be similar to $StepAdd1$ and $StepAdd2$.

We use the notation $\longrightarrow$ for representing a single step, and $\longrightarrow *$ for representing multiple steps. We rely on the substitution operation defined in Listing B.2 to handle functions. The

corresponding Coq code is stated in Listing B.3. Note how it maps almost directly the corresponding inference rules from Figure 3.2.

In the step rules definition, we use t (or t_1, t_2,..., t_n) for a generic term, and v (or v_1, v_2,..., v_n) for a generic value.

$$\frac{value\ v}{\lambda x : T_1, t\ v \longrightarrow \left[x := v\right]\ t}\ \textsc{StepAppAbs} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \textsc{StepApp1}$$

$$\frac{value\ v \qquad t_2 \longrightarrow t_2'}{v\ t_2 \longrightarrow v\ t_2'}\ \textsc{StepApp2}$$

(a) Step relation for functions: StepAppAbs; and function application: StepApp1, StepApp2

$$\frac{}{\left(\ if\ true\ t_1\ else\ t_2\ \right) \longrightarrow t_1}\ \textsc{StepIfTrue}$$

$$\frac{}{\left(\ if\ false\ t_1\ else\ t_2\right) \longrightarrow t_2}\ \textsc{StepIfFalse}$$

$$\frac{t \longrightarrow t'}{\left(\ if\ t\ t_1\ else\ t_2\right) \longrightarrow \left(\ if\ t'\ t_1\ else\ t_2\right)}\ \textsc{StepIf}$$

(b) Step relation for if: StepIfTrue, StepIfFalse, StepIf

Fig. 3.2. Reduction rules for functions, function application, and conditional terms.

Note that StepRec and StepRecAbs represent recursive functions. We use these relations to support for loops. Namely, *Verifloq* handles loops using recursive functions.

Step rules for typed lists can be found in Figure 3.4. Rules StepCons1, StepCons2, StepLcase1 handle the evaluation of subterms, StepLcaseNil handles the presence of a nil element and StepLcaseCons handles cases with values.

In Figure 3.5, a pair $\left(t_1, t_2\right)$, or tuple, is a structure with two terms with possibly different types. The term on the left is referred to as fst (first), and the term on the right is referred to as *snd* (second). The rules StepPair1, StepPair2, StepFst1 and StepSnd1 show the term evaluations

$$\frac{t_1 \longrightarrow t_1'}{add\ t_1\ t_2 \longrightarrow add\ t_1'\ t_2}\ \textsc{StepAdd1} \qquad \frac{value\ v \qquad t_2 \longrightarrow t_2'}{add\ v\ t_2 \longrightarrow cons\ v\ t_2'}\ \textsc{StepAdd2}$$

(a) Step relation for arithmetic operations (addition): StepAdd1, StepAdd2

$$\frac{t_1 \longrightarrow t_1'}{rec\ t_1 \longrightarrow rec\ t_1'}\ \textsc{StepRec}$$

$$\frac{}{rec\ (\lambda x : T \cdot t) \longrightarrow [x := (\ rec\ (\lambda x : T \cdot t\ )\ )]}\ \textsc{StepRecAbs}$$

(b) Step relation for recursive functions: StepRec, StepRecAbs.

Fig. 3.3. Reduction rules for boolean terms, natural numbers and binary operations.

$$\frac{t_1 \longrightarrow t_1'}{cons\ t_1\ t_2 \longrightarrow cons\ t_1'\ t_2}\ \textsc{StepCons1}$$

$$\frac{value\ v \qquad t_2 \longrightarrow t_2'}{cons\ v\ t_2 \longrightarrow cons\ v\ t_2'}\ \textsc{StepCons2}$$

$$\frac{t_1 \longrightarrow t_1'}{lcase\ t_1\ t_2\ x_1\ x_2\ t_3 \longrightarrow lcase\ t_1'\ t_2\ x_1\ x_2\ t_3}\ \textsc{StepLcase1}$$

$$\frac{}{lcase\ (\ nil\ T)\ t_2\ x_1\ x_2\ t_3 \longrightarrow t_2}\ \textsc{StepLcaseNil}$$

$$\frac{value\ v_1 \qquad value\ v_2}{lcase\ (cons\ v_1\ v_2)\ t_2\ x_1\ x_2\ t_3 \longrightarrow [x_2 := v_2\ ([x_1 := v_1]\ t_3)]}\ \textsc{StepLcaseCons}$$

Fig. 3.4. Step relation for lists: StepCons1, StepCons2, StepLcase1, StepLcaseNil, StepLcaseCons.

in pairs, whereas StepFstVal, StepSndVal represent how to extract the final values of a pair. As seen in Listing 3.5, tuples are used to store input and output wires.

Records follow the same idea, but with n-ary elements. Rules StepRecord, StepRecordProj1 and StepRecordProj2 show the step relation for records. These rules are defined in Figure 3.6.

$$\frac{t_1 \longrightarrow t_1'}{(t_1,\ t_2) \longrightarrow (t_1',\ t_2)}\ \textsc{StepPair1} \qquad \frac{value\ v \qquad t \longrightarrow t'}{(v,\ t) \longrightarrow (v,\ t')}\ \textsc{StepPair2}$$

$$\frac{t \longrightarrow t'}{fst\ (t) \longrightarrow fst\ (t')}\ \textsc{StepFst1} \qquad \frac{value\ v_1 \qquad value\ v_2}{fst\ (v_1,\ v_2) \longrightarrow v_1}\ \textsc{StepFstVal}$$

$$\frac{t \longrightarrow t'}{snd\ (t) \longrightarrow snd\ (t')}\ \textsc{StepSnd1} \qquad \frac{value\ v_1 \qquad value\ v_2}{snd\ (v_1,\ v_2) \longrightarrow v_2}\ \textsc{StepSndVal}$$

Fig. 3.5. Step relation for tuples: StepPair1, StepPair2, StepFst1, StepFstVal, StepSnd1 , StepSndVal

$$\frac{t_i \longrightarrow t_i'}{\{i_1 = v_1, \ldots, i_n = t_i\} \longrightarrow \{i_1 = v_1, \ldots, i_n = t_i'\}}\ \textsc{StepRecord}$$

$$\frac{t_0 \longrightarrow t_0'}{t_0 \cdot i \longrightarrow t_0' \cdot i}\ \textsc{StepRecordProj1} \qquad \frac{value\ v_i}{\{, \ldots, i = v_i, \ldots\} \longrightarrow v_i}\ \textsc{StepRecordProj2}$$

Fig. 3.6. Step relation for records: StepRecord, StepRecordProj1, StepRecordProj2.

Sum types, or disjoint unions, describe a set of values drawn from one of two given types. For example, *Nat* + *List Nat* indicates that a term can be of one of these two types. *inl Nat* + *List Nat* returns the left element, *Nat*, and *inr Nat* + *List Nat* returns the right element, *List Nat*. Namely, *inl* and *inr* extract the types, such as rules StepInl and StepInr demonstrate. Rules StepCase, StepCaseInl and StepCaseInr show the term evaluations in sums. These rules are defined in Figure 3.7.

Variants are a generalization of sum types for n-ary types. We use variant types to compute the type of a module taking into account the clock cycles. In our implementation, modules are treated as variant types, giving us a typed clocked behavior in our semantics. We describe further details in Section 3.3.1.

In Figure 3.8, we find the rules StepCaseEAsgn, StepCaseAsgn, StepBlocking and StepNonBlocking for the evaluation of blocking and non-blocking assignments. Rules StepModule, StepModuleMerge, StepModuleCommit and StepModuleEnd show the step relation for modules.

$$\frac{t_1 \longrightarrow t_1'}{inl\ T\ t_1 \longrightarrow t_1'}\ \textsc{StepInl} \qquad \frac{t_1 \longrightarrow t_1'}{inr\ T\ t_1 \longrightarrow t_1'}\ \textsc{StepInr}$$

$$\frac{t \longrightarrow t'}{case\ \left(t\ x_1\ t_1\ x_2\ t_2\right) \longrightarrow case\ \left(t'\ x_1\ t_1\ x_2\ t_2\right)}\ \textsc{StepCase}$$

$$\frac{value\ v}{case\ \left(\left(inl\ T\ v\right) x_1\ t_1\ x_2\ t_2\right) \longrightarrow \left[x_1 := v\right] t_1}\ \textsc{StepCaseInl}$$

$$\frac{value\ v}{case\ \left(\left(inr\ T\ v\right) x_1\ t_1\ x_2\ t_2\right) \longrightarrow \left[x_2 := v\right] t_2}\ \textsc{StepCaseInr}$$

Fig. 3.7. Step relation for sums: StepInl, StepInr, StepCase, StepCaseInl, StepCaseInr

The corresponding Coq code stating the step relation for sums, tuples, records, lists, assignments and module terms is defined in Listing B.3.

3.3 Typing Rules

As stated in Section 2.6, typing relations establish well-typed terms. Typing rules enable a strong type system for *Verifloq*, removing issues caused by Verilog's original weak type system. Figure 3.9 defines the straightforward typing relation for function abstractions, function application, recursion, natural numbers and boolean terms. Some trivial rules for binary and arithmetic operations are omitted for brevity. Figure 3.10 states the typing rules for lists, and Figure 3.11 defines the typing relation for pairs and records. Finally, Figure 3.12 defines the typing rules for sums and assignments. The corresponding Coq code is defined in Listing B.4.

$$\frac{value\ v_i}{case\ \left(\langle l_i = v_i\rangle\ as\ T\right)\ of\ \langle l_i = x_i\rangle\ t_i\ \longrightarrow\ \left[x_i := v_i\right] t_i}\ \textsc{StepCaseEAsgn}$$

$$\frac{t_i \longrightarrow t_i'}{case\ \left(\langle l_i = v_i\rangle\ as\ T\right)\ of\ \langle l_i = x_i\rangle\ t_i \longrightarrow case\ \left(\langle l_i = v_i\rangle\ as\ T\right)\ of\ \langle l_i = x_i\rangle\ t_i'}\ \textsc{StepCaseAsgn}$$

$$\frac{t_1 \longrightarrow t_1'}{blocking\ t_1\ t_2 \longrightarrow blocking\ t_1'\ t_2}\ \textsc{StepBlocking}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{non\_blocking}\ t_1\ t_2 \longrightarrow \text{non\_blocking}\ t_1'\ t_2}\ \textsc{StepNonBlocking}$$

(a) Step relation for assignments: StepCaseEAsgn, StepCaseAsgn, StepBlocking, StepNonBlocking.

$$\frac{t_1 \longrightarrow t_1'}{module\ l_1\ l_2\ t_1\ t_2 \longrightarrow module\ l_1\ l_2\ t_1'\ t_1'\ t_2}\ \textsc{StepModule}$$

$$\frac{value\ v_i \qquad t \longrightarrow t'}{module\ l_1\ l_2\ v_i\ t_1\ \longrightarrow module\ l_1\ l_2\ v_i\ t_1'}\ \textsc{StepModuleMerge}$$

$$\frac{value\ v_1 \qquad t_2 \longrightarrow value\ v_2}{module\ l_1\ l_2\ v_1\ t_2\ \longrightarrow module\ l_1\ l_2\ v_1\ v_2}\ \textsc{StepModuleCommit}$$

$$\frac{value\ v}{module\ l_1\ l_2\ v_1\ v_2\ \longrightarrow\ v}\ \textsc{StepModuleEnd}$$

(b) Step relation for modules: StepModule, StepModuleMerge, StepModuleCommit, StepModuleEnd.

Fig. 3.8. Reduction rules for module terms.

We keep the notation t (or t_1, t_2,..., t_n) for a generic term, and v (or v_1, v_2,..., v_n) for a generic value; n represents natural numbers. The prefir TR denotes an inference rule for *Typing Rule*.

Note that TRRec, the type of recursive functions, is simply a special case of function application, mapping a function to itself.

Lists carry elements of the same type, as rules TRNil, TRCons, TRLcase in Figure 3.10 show.

$$\frac{\Gamma\left(x\right) = T}{\Gamma \vdash x : T}\ \textsc{TRVar} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 \cdot t : T_1 \to T_2}\ \textsc{TRAbs}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 t_2 : T_1}\ \textsc{TRApp} \qquad \frac{}{\Gamma \vdash\ true : Bool}\ \textsc{TRTrue}$$

$$\frac{}{\Gamma \vdash\ false : Bool}\ \textsc{TRFalse}$$

(a) Typing rules for variables TRVar, functions TRAbs, function application TRApp, booleans TRTrue TRFalse.

$$\frac{\Gamma \vdash\ t_1 : Bool \qquad \Gamma \vdash\ t_2 : T \qquad \Gamma \vdash\ t_3 : T}{\Gamma \vdash\ if\ t_1\ t_2\ else\ t_3 : T}\ \textsc{TRIf} \qquad \frac{}{\Gamma \vdash\ n : Nat}\ \textsc{TRNat}$$

$$\frac{}{\Gamma \vdash\ unit : Unit}\ \textsc{TRUnit} \qquad \frac{\Gamma \vdash\ t_1 : T_1 \to T_1}{\Gamma \vdash\ rec\ t_1 : T_1}\ \textsc{TRRec}$$

(b) Typing rules for if statement TRIf, natural numbers TRNat, unit type TRUnit, recursion TRRec.

Fig. 3.9. Typing rules for functions, function application, recursion, natural numbers and boolean terms.

$$\frac{}{\Gamma \vdash\ nil\ T_1 : List\ T_1}\ \textsc{TRNil} \qquad \frac{\Gamma \vdash\ t_1 : T_1 \qquad \Gamma \vdash\ t_2 : List\ T_1}{\Gamma \vdash\ cons\ t_1\ t_2 : List\ T_1}\ \textsc{TRCons}$$

$$\frac{\Gamma \vdash\ t_1 : List\ T_1 \qquad \Gamma \vdash\ t_2 : T_2 \qquad x_1 : T_1,\ x_2 : List\ T_1,\ \Gamma \vdash\ t_3 : T_2}{\Gamma \vdash\ \left(case\ t_1\ t_2\ x_1\ x_2\ t_3\right) : T_2}\ \textsc{TRLcase}$$

Fig. 3.10. Typing rules for lists: TRNil, TRCons, TRLcase.

Pairs, and more generally, records, aggregate terms of different types. The respective typing rules are described in Figure 3.11.

Sums, and more generally, variants, will have one particular type among other possible types. The corresponding typing rules are described in Figure 3.12.

$$\frac{\Gamma \vdash\ t_1 : T_1 \qquad \Gamma \vdash\ t_2 : T_2}{\Gamma \vdash\ (t_1,\ t_2) : (T1,\ T2)}\ \textsc{TRPair} \qquad \frac{\Gamma \vdash\ t : T_1 * T_2}{\Gamma \vdash\ fst\ t : T_1}\ \textsc{TRFst}$$

$$\frac{\Gamma \vdash\ t : T_1 * T_2}{\Gamma \vdash\ snd\ t : T2}\ \textsc{TRSnd}$$

(a) Typing rules for pairs: TRPair, TRFst, TRSnd.

$$\frac{\Gamma \vdash\ t_1 : T_1 \qquad \ldots \qquad \Gamma \vdash\ t_n : T_n}{\Gamma \vdash\ \{i_1 = t_1, \ldots, i_n = t_n\} : \{i_1 = T_1, \ldots, i_n = T_n\}}\ \textsc{TRRecord}$$

$$\frac{\Gamma \vdash\ t : \{\ldots, i : T_i, \ldots\}}{\Gamma \vdash\ t \cdot i : T_i}\ \textsc{TRProj}$$

(b) Typing rules for projections: TRRecord, TRProj.

Fig. 3.11. Typing rules for pairs and records.

$$\frac{\Gamma \vdash\ t_1 : T_1}{\Gamma \vdash\ inl\ T_2\ t_1 : T_1\ T_2}\ \textsc{TRInl} \qquad \frac{\Gamma \vdash\ t_2 : T_2}{\Gamma \vdash\ inr\ T_1\ t_2 : T_1\ T_2}\ \textsc{TRInr}$$

$$\frac{\Gamma \vdash\ t_0 : T_1\ T_2 \qquad \Gamma \vdash\ t_1 : T_1 \to T_3 \qquad \Gamma \vdash\ t_2 : T_2 \to T_3}{\Gamma \vdash\ case\ t_0\ x_1\ t_1\ x_2\ t_2 : T_3}\ \textsc{TRCase}$$

(a) Typing rules for sums: TRInl, TRInr, TRCase.

$$\frac{\Gamma \vdash\ t_j : T_j}{\Gamma \vdash\ \langle l_j = t_j \rangle : \langle l_i : T_i \rangle}\ \textsc{TRAsgn} \qquad \frac{\Gamma \vdash\ t_0 : \langle l_i : T_i \rangle \qquad \Gamma, x_i : T_i \vdash\ t_i : T}{\Gamma \vdash\ (\ case\ t_i\ x_i\ t\ ) : T}\ \textsc{TREAsgn}$$

(b) Typing rules for assignments in a module: TRAsgn, TREAsgn.

Fig. 3.12. Typing rules for sums and assignments.

Finally, the typing rules previously shown are not the same as *type checking*. Type checking consists of an algorithm with a function that tells whether of not a term is well-typed. It can be derived from typing rules, but it is not a requirement to prove type safety.

3.3.1 Adding Clock Behavior to Verifloq

At this stage, the current semantics for *Verifloq* does not take into account any sort of clocked evaluation for positive edge events. To address this issue, we adopt the ideas from [Lööw and Myreen 2019] (and its extended work, [Lööw 2021]). In this formalization, authors describe an alternative three-layer-style semantics for a Verilog-like language formalized in HOL4. The first layer evaluates expressions; the second layer steps a process. These are both unclocked phases, and are, to some extent, similar to our current implementation for *Verifloq* - the main difference is that *Verifloq* supports fewer language constructs.

The third layer of the semantics proposed by [Lööw and Myreen 2019] presents of a clocked evaluation function, mrun, that steps a module forward given a specified number of cycles. An intermediate function produces a new initial state for a respective new cycle and merges the non-blocking statements with the program variables (blocking evaluated expressions). We use this core idea for implementing the step relation for modules in *Verifloq*, and we limit the clock cycles to one. In Listing 3.7, the function mstep_merge recursively evaluates blocking expressions, then non-blocking expressions and merges them in a deterministic queue, enabling the clocked evaluation of the module.

```
Inductive step : tm -> tm -> Prop :=
  (* Previously defined step relations above... *)
  | Step_Module : forall t1 t2 t1' l1 l2
      t1 ⟶ t1' ->
      tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 t1' t2
  | Step_ModuleMerge : forall t1 t2 t1' l1 l2
      val v=(mstep_merge l1 l2 t1 t2) ->
```

```
      t1 ⟶ t1' ->
      tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 v t1'
  | Step_ModuleCommit : forall t1 t2 t1' l1 l2 v1
      val v1 ->
      t2 ⟶ val v2=(mstep_commit l1 l2 t1 t2) ->
      tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 v1 v2
  | Step_ModuleEnd : forall t1 t2 t1' l1 l2 v
      val v ->
      tm_module l1 l2 t1 t2 ⟶ v
```

Listing 3.7. Step relation for modules.

3.3.2 Typing with Clock Behavior

Recall Section 3.1.1, where we specify the terms translation in *Verifloq*.

```
( tm_module (tm_var "HalfAdd") (tm_const 0)
  (tm_record
    (tm_tuple [ (tm_input "a"), (tm_input "b") ],
      [ (tm_output "sum"), (tm_output "carry") ]
    )
    sum :::= (tm_xor a b)
    carry :::= (tm_and a b)
  )
  []
)
```

Listing 3.8. Half-adder example translated to *Verifloq*'s terms.

Note that in Listing 3.8, the second argument of the module represents the clock. The last argument represents the list of blocking and non-blocking assignment. Throughout the time execution, we navigate the module states by assigning different disjoint union types to it, such that $module : T_1 + T_2 + ... + T_i$, where i is the number of cycles specified as an input for the execution. Figure 3.13 exemplifies the idea for 3 clock cycles.

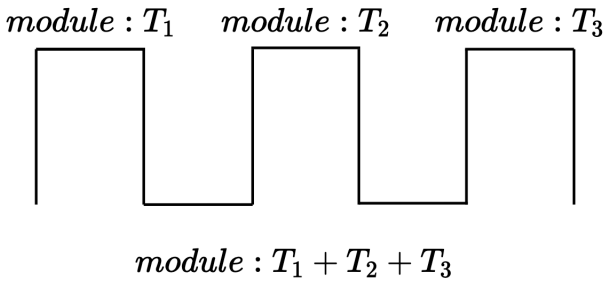

Fig. 3.13. Module types throughout clock increments.

Note that our semantic model substantially differs from [Lööw and Myreen 2019], where authors do not formalize a type system for Verilog. Instead, they check possible type errors at runtime.

3.4 Type Safety Guarantees

Verifloq's type system gives us an important guarantee: our language will never reach a *stuck state*. Stuck terms correspond to meaningless or erroneous programs [Pierce 2002]. In this section, we state and prove crucial two theorems to demonstrate that *Verifloq*'s evaluated programs are safe: *Progress and Preservation*. Our formalization choices keep these proofs simple and straightforward.

In Theorem 3.1, we state that closed, well-typed terms are either a value or can safely take a reduction step. Well-typed terms will always evaluate to a value or will follow a particular pre-established rule to be evaluated to another expression. In other words, well-typed terms

will never reach a *stuck state*. Stuck states characterize a situation where the semantic model does not know what to do because the program has reached an unexpected or meaningless state [Pierce 2004]. Theorem 3.1 is also know as **Progress**, and the Coq definition is stated in Listing B.6. The full development of the mechanized proof can be found on Appendix B.

THEOREM 3.1 (**PROGRESS**).

If t is a well-typed term, then either t is a value or there is some well-typed term t' with $t \longrightarrow t'$.

PROOF. By induction on terms, as shown in Listing B.6. It can also be done by induction on the derivation of $\vdash t : T$. □

We also prove the **Preservation** theorem for *Verifloq*, stated in Theorem 3.2. It ensures that if a well-typed term takes a step, the result will be also well-typed. The corresponding Coq mechanization is stated in Listing B.7.

THEOREM 3.2 (**PRESERVATION**).

If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

PROOF. By induction on the derivation of $\vdash t : T$, as stated in Listing B.7. We rely on an auxiliary theorem for showing the substitution also preserves types. This theorem is stated in Theorem 3.5. □

Once Theorem 3.1 and Theorem 3.2 are proved, we can say that *Verifloq* is type-safe with respect to the semantics we proposed.

3.4.1 Additional Guarantees

Although **Progress** and **Preservation** were the most important target guarantees to prove *Verifloq*'s' type safety, we can prove other properties of our language.

In Theorem 3.3, we state the theorem for type uniqueness, which says that each term t has at most one type. The Coq definition is defined in Listing B.5.

THEOREM 3.3 (TYPE UNIQUENESS).

If $\Gamma \vdash e : T$ *and* $\Gamma \vdash e : T'$*, then* $T = T'$.

PROOF. By induction on the derivation of $\vdash e : T$. The full Coq proof is described in Listing B.5. □

We also prove that we can safely add assumptions to our mapping function, the context Γ, without losing any other valid typing statements. This theorem, also known as *weakening* property, is stated in Theorem 3.4, and the corresponding Coq mechanization is declared in Listing B.9.

THEOREM 3.4 (WEAKENING).

If $\Gamma \vdash t : T$ *and* $x \notin \Gamma$*, then* $\Gamma, x : S \vdash t : T$.

PROOF. By induction on the typing derivations. □

In Theorem 3.5, we prove that our substitution operation preserves the typing relation. The Coq definition is defined in Listing B.8.

THEOREM 3.5 (SUBSTITUTION PRESERVES TYPES).

If $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash s : S$*, then* $\Gamma\ [x := s]\ t : T$.

PROOF. By induction on the derivation of the statement $\Gamma, x : S \vdash t : T$. □

3.5 Simple Verilog-like Programs with Verifloq

Verifloq allows the development of simple Verilog programs, which should be declared in traditional *.v* files. We also provide a parser to Coq. At this time, the parser is unverified.

Alternatively, *Verifloq* code can be stated directly into Coq files, since we also provide a translation using Coq's Notation command.

Besides Listing 3.4, previously presented, we also have Listing 3.9 and Listing 3.10 as examples of simple Verilog programs that match *Verifloq*'s supported constructs.

```
module counter (input clk, reset, output[3:0] counter);
    reg [3:0] counter_up;
    always_ff @(posedge clk)
    begin
      if(reset)
        counter_up <= 4'd0;
      else
        counter_up <= counter_up + 4'd1;
    end
    assign counter = counter_up;
endmodule
```

Listing 3.9. Simple counter.

```
module logical_operators (x1, x2, x3, z1, z2, z3, z4);
input [1:4] x1, x2, x3; output z1, z2, z3, z4;
assign z1 = (x1 && x2) && x3,
       z2 = (x1 || x2) && x3,
       z3 = (x1 && x3) || x2,
       z4 = !(x1 || x3);
endmodule
```

Listing 3.10. Basic logical operations.

3.6 Summary and Benefits of Verifloq

The verification techniques presented in Chapter 2 do not take into consideration issues caused by language design choices of the Verilog language itself. These issues are often not detected in compile time. A classic example is an if statement lacking an else clause, possibly unwantedly resulting in a latch [Chonnad and Balachander 2007] in some Verilog implementations. This issue happens due Verilog's weak type system, and *Verifloq* addresses the problem by delivering a stronger type system for a subset of constructs for the original Verilog language.

Concrete applications for *Verifloq* are described in Chapter 4, where we propose combining *Verifloq* with Hoare Logic. *Verifloq* can also act as an important component of a multi-staged hardware verification pipeline.

In *Verifloq*, we define a verified operational semantics, and we demonstrate it is a type-safe HDL by stating and proving Progress and Preservation theorems. *Verifloq* aims to deliver a work environment closer to the original Verilog language inside Coq. Other projects also propose semantic models to handle HDL or HLS, and Chapter 5 brings an extensive comparison with other existing verification models targeting language aspects of HDL.

The code reference can be found on Appendix A, and a summary of relevant Coq tactics and commands is listed on Appendix B.

3.6.1 Limitations

Verifloq is a functional implementation of a type-safe HDL, but it has limitations. First, although we implement a large set of language constructs, *Verifloq* does not support the complete language features from the original Verilog language at this time. Another extension that needs future work is the full verification of our current parser.

Also, we do not provide a type checking algorithm in this project. We work exclusively with typing rules to prove type safety. A type checking algorithm is a function that tells whether of not a term is well-typed.

Our development is limited to single modules and positive edge circuits. In fact, this limitation also happens in all other existing formalizations for HOL and Coq. Building a semantic model to handle all the concurrency aspects of HDL is an open research problem. Hence, designing a robust semantic model for Verilog is challenging, and Chapter 5 also examines distinct Verilog and general HDL semantic approaches.

We also do not investigate performance aspects in this thesis. Our goal is to provide a type-safe implementation. Future versions of *Verifloq* may target optimizations. Similarly, the formalization does not extend to synthesis. Section 6.2 discussed future work and other challenging open research questions.

4 Applications

Verifloq, described in Chapter 3, allows us to write simple Verilog programs, such as basic counters and adders. Initially, these implementations may sound boring, but since they are built in Coq, we can benefit from other features of the proof assistant. An immediate feature is verification with Hoare Logic (see [Hoare 1969] and [Pratt 1976]). By equipping each language construct with proof rules, we can check, in a compositional way, if programs satisfy an expected behavior. These proof rules mirror the structure of the program, making them a convenient way to verify correctness. Section 4.1 discusses this application in depth.

Another application for *Verifloq* is program equivalence. Determining equivalence of Verilog code can be useful for identifying, for example, if two High-Level Synthesis tools produce the same outputs. Whereas pure syntactic equivalence can be derived in a straightforward way, behavioral or contextual equivalence often need more work. Section 4.2 expands the discussion on program equivalence.

Lastly, Section 4.3 investigates the idea of *Verifloq* as a component of a multi-stage hardware verification pipeline.

4.1 Hoare Logic and Assertions

We often work with Hoare Logic by stating **Hoare Triples**. These are claims about the state of a piece of code, before and after its execution. Listing 4.1 shows the notation of a Hoare Triple. `{P}` is a statement that stands for *pre-condition*. `{Q}` stands for *post-condition*, which represents the expected output.

```
{P}code{Q}
```

Listing 4.1. Hoare Triple.

In the literature, {P} and {Q} are referred to as *assertions*. In the context of Hoare Logic, they have a distinct meaning from assertions we saw in Chapter 2. To avoid naming collisions, let us refer to assertions in the context of Hoare Triples as Hoare assertions. Hoare assertions, which will express pre and post-conditions, are often FOL formulas.

Every language construct receives its own proof rule. For instance, the proof rule for an if clause is specified in Figure 4.1.

$$\frac{\vdash \{P \wedge b\}\ S_1\ \{Q\} \qquad \vdash \{P \wedge \neg b\}\ S_2\ \{Q\}}{\vdash \{P\}\ \text{if}\ b\ S_1\ \text{else}\ S_2\ \{Q\}}\ \text{if}$$

Fig. 4.1. Proof rule for an if clause.

In *Verifloq*, we implement rules for a reduced set of language constructs as a proof of concept for a small study case in this section. These rules can be found in the applications/ ↪ hoare.v file in the repository. We currently support if/else and assign and while constructs. The rule for if was stated in Figure 4.1; Figure 4.2 shows the proof rules for assign and while.

$$\frac{}{\vdash \{[\ x := e\ ]\ Q\}\ x := e\ \{Q\}}\ \text{assign} \qquad \frac{\vdash \{P \wedge b\}\ \text{c}\ \{P\}}{\vdash \{P\}\ \text{while}\ b\ \text{do}\ c\ \{P \wedge \neg b\}}\ \text{while}$$

Fig. 4.2. Proof rule for assign and while loops.

We also verify these rules in Coq. In Listing 4.2, we prove the assign construct and provide an example of a valid, verified Hoare triple.

```
Theorem hoare_assign : forall Q X e,
  {Q (mapsto X e)} X := e {Q}.
Proof.
  unfold hoare_triple.
  intros Q X e st st' HE HQ.
  inversion HE. subst.
  unfold assn_sub in HQ. assumption.
Qed.

Theorem hoare_e1 :
  {True} X := 2 {X = 2}.
Proof.
  eapply conseq_pre.
  - apply hoare_asgn.
  - auto.
Qed.
```

Listing 4.2. Proof of assign rule and corresponding Hoare triple example.

In Listing 4.3, we provide an example of a valid, verified Hoare triple for an if/else clause.

```
Theorem hoare_if : forall P Q b c1 c2,
  {P ∧ b} c1 {Q} ->
  {P ∧ ¬ b} c2 {Q} ->
  {P} if b then c1 else c2 end {Q}.
Proof.
  intros P Q b c1 c2 HTrue HFalse st st' HE HP.
```

```
  inversion HE; subst; eauto.
Qed.


Theorem hoare_e2 :
    {True}
  if (X = 0)
    Y := 10
    else Y := X
  end
    {X <= Y}.
Proof.
  apply hoare_if.
  - eapply conseq_pre.
    -- apply hoare_asgn.
    -- assn_auto.
      simpl. intros st [_ H].
      apply eqb_eq in H.
      rewrite H. lia.
  - eapply conseq_pre.
    -- apply hoare_asgn.
    -- assn_auto.
Qed.
```

Listing 4.3. Proof of if/else rule and corresponding example.

It is also possible, in a relatively simple way, to extend the supported constructs using ideas of open-source Hoare libraries. The Sail project [5], for example, offers a robust set of verified rules for multiple constructs [6].

Hoare-style verification statically proves that, given a precondition, a particular post-condition will hold after a piece of code executes. We generate a logical formula, the *verification condition*, that demonstrates the program behaves as specified if it becomes true.

To some extent, Hoare Triples resemble Assertion Based Verification (ABV) (covered in Section 2.2). However, instead of an automaton checker, we have an axiomatic semantic model for each language construct. The two verification methods can be complementary in a verification pipeline, as shown in Figure 4.3.

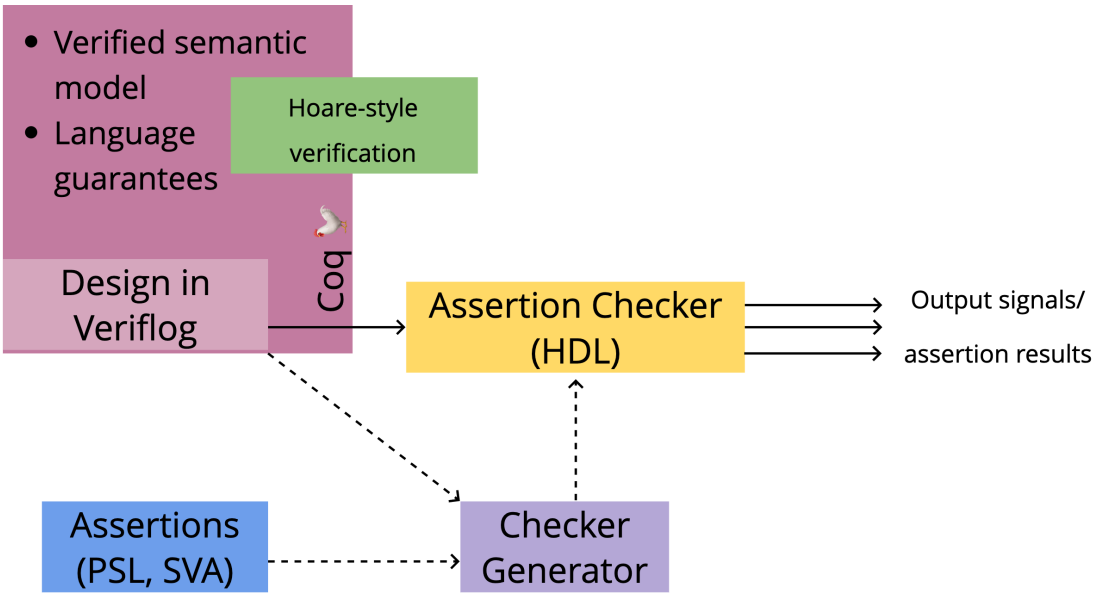

Fig. 4.3. Verification pipeline with Verifloq, Hoare Triples and ABV.

[5] https://github.com/rems-project/sail
[6] https://github.com/rems-project/sail/blob/37247bb1767015bfee35d9536040d83cf151339f/lib/coq/Hoare.v

In summary, the Hoare-verification style provides an extra layer of verification benefiting from Coq's settings. It can be seen as a language on top of *Verifloq* aiming to assert and prove properties of programs. The proof rules we establish play the role of an assertion checker in ABV, such as MBAC [Boulé and Zilic 2005]. We can prove the assertions and also obtain a symbolic method for deriving them. This final asset gives margin to extend the verification pipeline to SMT solvers, for example.

4.2 Program Equivalence for HLS

In High-level Synthesis (HLS), hardware is described in a higher-level language (HLL), such as C or C++, and then it is translated to a traditional HDL, such as Verilog or VHDL ([Coussy and Morawiec 2008]). HLS enables teams with a stronger software background to deliver hardware-oriented projects. It also provides an "easy way to test the performance of the algorithmic flow by promptly prototyping the hardware design and testing it on FPGAs" [Pundir et al. 2021]. Some vendors, such as Vivado HLS [7], offer a robust integrated environment with the ability to generate optimized HDL. However, there are multiple known issues with HLS. The skill and technical backgrounds associated with hardware and software coding practices are essentially different, and can result in vulnerable HLS-generated RTL. Moreover, translating these paradigms - HLL to HDL - can be challenging. For example, is there a meaning for C pointers in HDL? At this time, those are open research questions.

HLS tools also cannot always guarantee that the hardware designs they produce are equivalent to the input software specifications they were given. Projects like *Vericert* [Herklotz et al. 2021], as well as the work proposed by [Mathur et al. 2009], try to undermine

[7] https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html

this issue of software-to-hardware equivalence. [Herklotz et al. 2021] uses Compcert [Leroy 2009] for its HLL. Compcert is a verified C compiler, and *Vericert* implements the semantic model proposed by [Lööw and Myreen 2019] to produce a verified Verilog output. In [Mathur et al. 2009], authors describe the strategy of Sequential Equivalence Checking (SEC). SEC checks the equivalence of two distinct designs, even when there is no one-to-one correspondence between their state elements. Authors use SEC to establish a workflow to enable functional equivalence check between HLLs and generated HDLs.

Besides the HLL-to-HDL equivalence, there seems to be another important open question: what can we say about the equivalence between two or more generated HDLs? Namely, if we input the same C program to Vivado HLS and another vendor, such as LegUp[8] [Canis et al. 2011], can we guarantee their output HDL are equivalent? Program equivalence is a vast research area, and *Verifloq* can be a key component prove this kind of equivalence.

There are different definitions of equivalence we can consider in program equivalence. In behavioral equivalence, we want to determine if expressions evaluate to the result in every state [Simpson and Voorneveld 2020]. In contextual equivalence, we determine if two program expressions are equivalent if any occurrences of the first expression in a complete program can be replaced with the second one without affecting the observable results of the program execution. In [Pitts 2000] and, more extensively, in [Pitts 1997], the author presents a collection of techniques to design or enhance an operational semantics such that contextual equivalence of programs can be demonstrated.

Program equivalence is a wide research field, and while specific techniques are out of the scope of this work, we can envision an extension for *Verifloq* where we can demonstrate program equivalence (behavioral or contextual) upon extension of our existing operational semantics. Since *Verifloq* handles pure Verilog programs, it would be possible to design

[8] http://legup.eecg.utoronto.ca/

a pipeline that receives different output HDL from HLS tools, and produces proofs of equivalence for the generated Verilog programs. This pipeline would be an interesting case study to establish a comparison between distinct vendors.

4.3 Towards a Correct-by-construct Verified Hardware Pipeline

In Chapter 2, we became familiar with a wide range of verification options for HDL. In many cases, these strategies can be complementary. For example, it is possible to combine the idea of test-benches with ABV [Bombieri et al. 2006]. Figure 4.4 illustrates this use case.

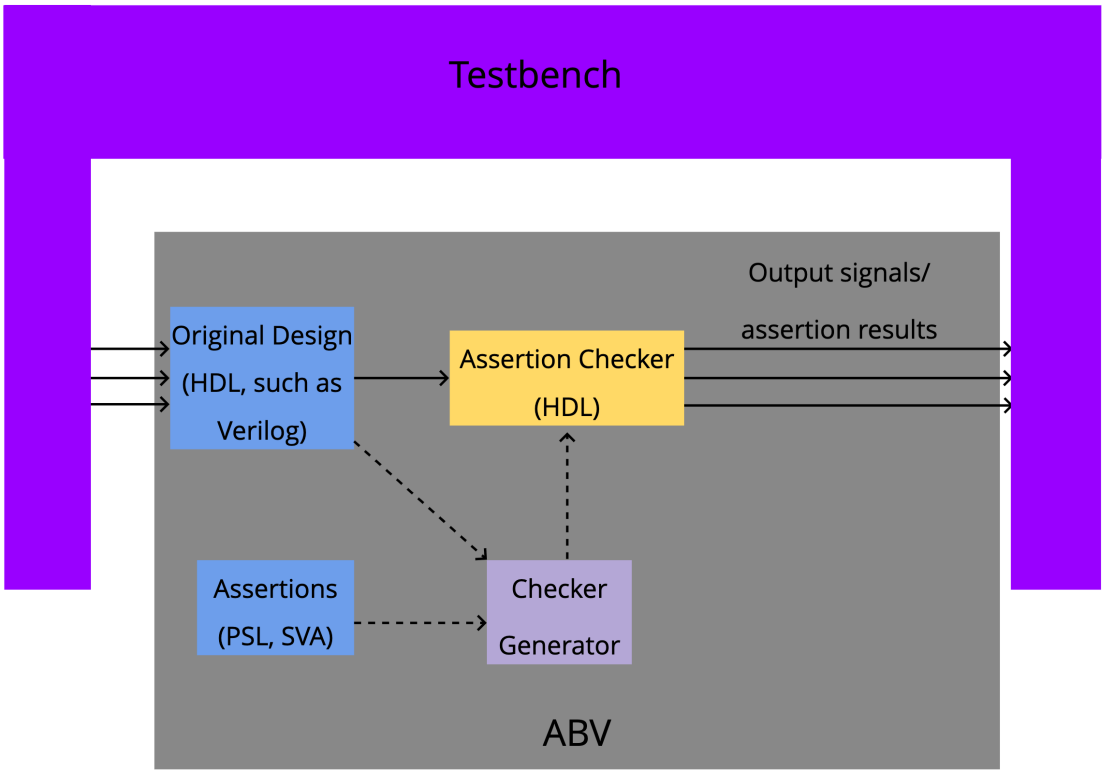

Fig. 4.4. ABV and test-bench techniques combined.

Verifloq can contribute with an extra layer of verification in the language aspect, in a correct-by-construct oriented way. Incorporating the ideas of Figure 4.3 and Figure 4.4, we can compose a more robust verification pipeline in Figure 4.5. This enhanced pipeline gives us guarantees in different phases of the design.

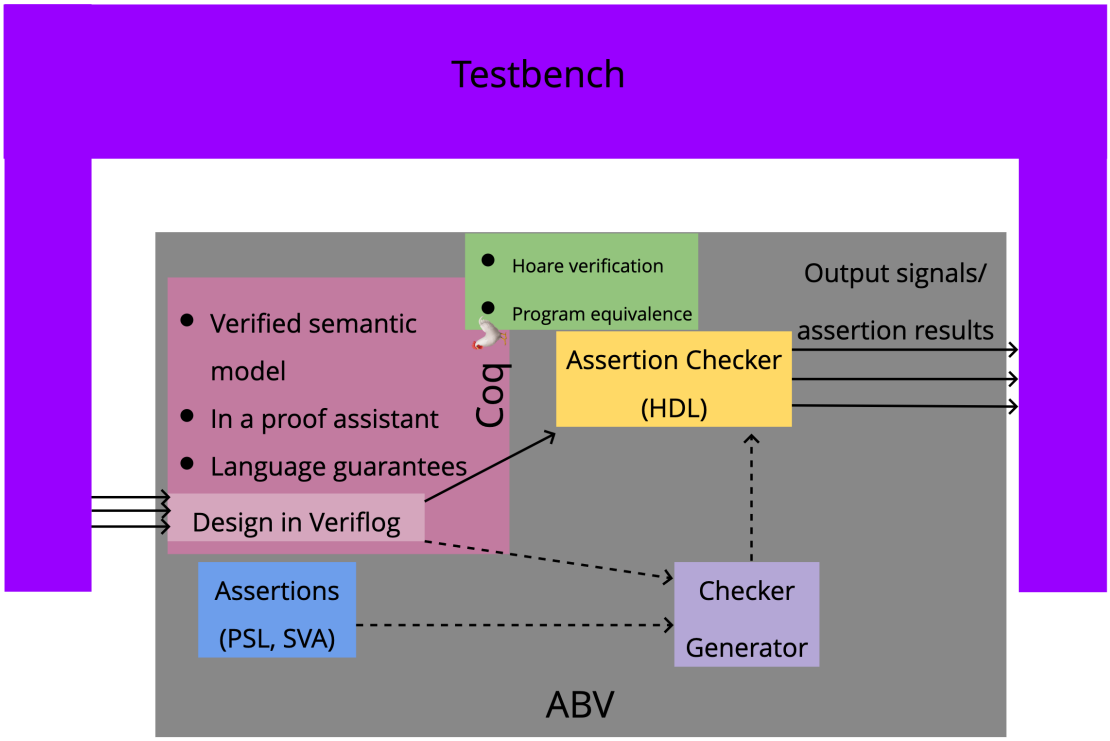

Fig. 4.5. Verification pipeline incorporating Verifloq.

5 Comparison: Alternative HDL Formalizations

Chapter 4 presented versatile use cases for *Verifloq*, and proposed its integration with other existing verification techniques. In this section, we provide a deeper discussion on *Verifloq*'s semantic model, and compare our choice with alternative formalizations.

5.1 VHDL vs. Verilog vs. *Verifloq*

Other HDL, such as VHDL, also have strongly typed systems. VHDL follows the Ada/-Pascal style, and it is a relatively verbose language when compared to Verilog's constructs [Bailey 2003]. The language structure also makes the development of a small-step operational semantics challenging. In [Golson et al. 1994], authors show how a Finite State Machine (FSM) model for VHDL is hard to be achieved. Specifically, issues arise with vendor-specific extensions (which modify the entire meaning of language constructs) and arbitrary encodings (declared as enumerated types), which are hard to be formalized as states.

Semantic models for HDL formalizations have been an active research topic for the past four decades. [Kloos and Breuer 1995] presents a collection of techniques to define the formal semantics of VHDL. Among them, the book presents a semantics for VHDL based on Petri nets. A Petri net is a graph model representing the behavior of systems exhibiting concurrency in their operation, and it also carries information about the transitions of the system. A Petri net represents a type of nondeterministic state machine, but in a convenient form for modeling and analyzing concurrent systems [Dennis 2011]. This representation

style fits VHDL's concurrent nature. Authors successfully cover a large set of VHDL constructs in their formalization, but their execution model becomes very complex and difficult to be formally verified by a theorem prover. Moreover, there is no clear indication the Petri nets approach could be easily attached to model checkers or theorem provers.

Another semantic model for VHDL described in the same book introduces a deterministic model using finite state machines (FSM). Authors elaborate on a series of finite automata in which transitions are labelled with conditions and transformations. The states of this automaton denote control flow nodes within the execution model of VHDL, and data (typed variables) is handled in a separate space. There are dedicated automata for sequential statements and processes. This approach turns out to be very compositional, and it can be linked to proof systems. However, the immediate limitation of this work is the specification of requirements, which is tightly bound to temporal logic. Users cannot specify anything about resources, for instance.

[Kloos and Breuer 1995] also presents an operational semantics for a subset of VHDL, named *Femto-VHDL*. Booleans, natural numbers, sequential statements, signal assignments, and concurrent statements are formalized in the model. Inference rules describe a state system for a complete simulation loop. Authors aimed to deliver a model entirely dissociated from any particular proof assistant, as opposed to [Umbreit 1992], which, in contrast, has a VHDL model bound to the LAMBDA proof assistant [McIsaac 1993]. Although very robust, this model has proven to be difficult to extend, given the numerous rules for state manipulation. It also requires the user to specify complex initial signal interrelationships in order to keep the analysis from degenerating into an non-exhaustive simulation.

In [Goossens 1995], authors propose a combined operational and observational semantics that includes the notions of delta time, zero-delay scheduling, arbitrary wait statements, and resolution functions for VHDL. This work innovatively adapts the idea of bisimulation

to hardware settings. Bisimulation is a binary relation on the terms of a language that is invariant under the observable states of the language. In other words, it describes behaviors that can be observed out of the terms [Pous and Sangiorgi 2019]. Its core idea has mostly emerged from Concurrency Theory, and it potentially fits the HDL scenario, which also requires reasoning about concurrent states. However, the semantics proposed by [Goossens 1995] is very restricted and does not support very basic language constructs, such as functions.

As seen in Section 2.5, a semantic model is required to formally establish type safety guarantees for a language. Unfortunately, we lack an extensible operational semantics for VHDL, and hence, we are not able to provide the typing guarantees we achieve with *Verifloq*. In summary, although VHDL is also a strongly typed language, we do not have a current model that allows us to formally prove its type safety properties.

Verilog, on the other hand, has language constructs more similar to C. The current research scenario provides a well-grounded knowledge base to develop an operational semantics for this kind of language. Nevertheless, the formalization of Verilog has also been a last-long challenge. In the majority of the cases, including this document, a subset of the original language is adopted. In [Gordon 1995], authors formalize a syntactic subset of Verilog in a language called *V*, which is very similar to the one we have for *Verifloq*. They support multiple modules by performing a flattening process (*normalization*) in the program, where all the modules are condensed in a single one. A similar strategy is also present in HLS tools. Their semantics is based on a state representation of simulating clock cycles.

This simplified semantic model has been adopted in multiple formalizations, such as [Meredith et al. 2010], where authors develop a much more rigorous mathematical model for Verilog's behavior. In [Dimitrov 2001], we find a semantic model considering the parallel

aspects of much larger subset of Verilog. In [Yongjian and Jifeng 2003], authors use the semantics of *V* language as the baseline for a theory of bisimulation for Verilog. [Gordon 1995] delivers a compact, but robust formalization of a subset of Verilog, expanding the original set proposed in the *V* language. However, [Gordon 1995] gives a strong emphasis to simulation cycles, and therefore does not provide a structured or systematic way to expand its syntax. In other words, an expansion "recipe", such as the one we presented in *Verifloq* on Chapter 3, is not possible.

In all the aforementioned projects, there is no emphasis on typing guarantees, so none of the proposed models for Verilog claim type-safety properties. These models do not address the issue of Verilog's weak type system. In contrast, *Verifloq* not only proposes an operational semantics for a subset of Verilog's constructs, but it also provides appropriate typing rules to ensure type safety guarantees.

5.1.1 Benefits of Verifloq's Operational Semantics

In *Verifloq*, we propose a small-step operational semantics. The immediate benefit of this approach, as seen in Section 5.1, is that we can prove properties about the execution of programs. Another advantage of this semantic style is the direct application in program equivalence, as stated in Section 4.2. All the approaches described in [Pitts 2000] rely on operational semantics for introducing proofs of contextual equivalence between programs.

A small-step operational semantics provides a concrete "recipe" to expand language features when it is fully mechanized in Coq. In Chapter 3, we describe the core structure of the *Verifloq*, and if we need to add another construct, we know we must follow the following steps in the formalization:

(1) If the construct introduces new types (which can be added by our design choice), state the type definition inductively;

(2) State the terms;

(3) Introduce the translation notation;

(4) If the construct introduces new values, add their inductive definitions;

(5) Adjust the substitution operation accordingly;

(6) Add the corresponding step relation;

(7) Define the required typing rules;

(8) Ensure the proofs of progress and preservation hold.

Some of these steps may be non-trivial, i.e. introducing an appropriate step rule, but the approach we take in *Verifloq* presents a consistent method to give us a systematic approach to work on.

Another advantage of building *Verifloq* in Coq is that every time we modify the semantics, the proof assistant will let us know if the proofs of progress and preservation still hold. If not, we will know immediately that our proofs need adjustments to support the new constructs or eventual modifications, and we can catch issues if changes break the safety of the semantic model.

5.2 Verifloq and HLS Formalizations

In Section 4.2, we saw a brief introduction to HLS. Recall that in HLS, users can describe the circuit design in a higher-level specification language, such as C or C++, and have their outcomes compiled to an RTL model. Vivado HLS [9] and the Intel HLS Compiler [10] are examples of widely adopted HLS tools.

[9] https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html

[10] https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

Vericert [Herklotz et al. 2021] is a formally verified HLS asset also built on the Coq proof assistant. Similarly, other HLS projects have taken advantage of theorem provers to provide a verified output design that has the same behavior as its input program. Kôika [Bourgeat et al. 2020] targets a subset of the Bluespec [11] [Nikhil 2004] language, also using Coq. [Reynolds et al. 2019] introduces a Coq formalization for ReWire, an HDL based in Haskell which produces VHDL code. Authors use a monadic approach to represent time and effects. Finally, [Flor et al. 2015] presents Π-ware, a unified typed language for the design, simulation, verification, and synthesis of hardware circuits built on top of Agda[12] [Norell 2007].

In all the HLS projects previously mentioned, there is an emphasis on HLL. These projects control the generated Verilog (equivalent HDL code) after processing the inputs from another HLL. In *Verifloq*, however, we have a formalization targeting Verilog, and not another higher-level language. Currently, we aim to provide the users with a tool where designers can write Verilog and, assuming their code is contained in our set of supported constructs, then they can automatically have type-safe guarantees from the HDL. Users can also benefit from a theorem prover environment if they need to write their own proofs. In this sense, *Verifloq* does not target HLS, but rather proposes a safe HDL, with a safe semantics verified in Coq. This is achieved by *Verifloq*'s strong type system, enforced by the typing rules specified in Chapter 3.

Lastly, [Lööw and Myreen 2019] proposes a semantic model for Verilog that does not take into account any types, and their implementation checks possible type errors at runtime. *Verifloq*, on the other hand, offers guarantees at the type-level.

[11] https://bluespec.com/
[12] https://github.com/agda/agda

6 Conclusion

This thesis addressed the need for a verified, type-safe language that can rule out undesired faults in hardware projects due to language issues. We presented and implemented *Verifloq*, a strongly typed HDL based on a subset of the original Verilog language. Often neglected in the hardware verification domain, faults generated by language design issues may happen, and unverified languages should not be taken as an absolute source of truth. Circuit designers should be able to detect these issues, in the same way they are able to verify other design faults.

Verifloq uses the Simply-Typed Lambda Calculus as its baseline, and we demonstrate how to expand the language constructs in the STLC, so they match Verilog's syntax. We developed a flexible small-step operational semantics for our language, and combined with its set of typing rules, we proved *Verifloq*'s type safety in the Coq proof assistant. Although both *Verifloq* and VHDL are strongly typed HDL, *Verifloq* has a verified type system, and with the proofs of progress and preservation, we guarantee that well-typed terms can never reach a stuck state during evaluation. We also present additional typing guarantees, such as type uniqueness.

Our project differs from High-level Synthesis (HLS) because our core object of study is the formalization of the HDL component, and not a Higher-Level Language (HLL). The latter will produce HDL (pure Verilog or VHDL), which is then, once again unverified. In addition, *Verifloq* extends the verification pipeline to prevent design faults by adding an extra layer of safety at the language level.

Verifloq has a wide range of applications, and we demonstrate how it can be used with other formal verification strategies. In particular, we develop a use case for Hoare Logic,

where we can easily write and prove assertions about simple Verifloq programs. We also show how our implementation can be used in combination with HLS to prove program equivalence of generated HDL. Finally, we also propose multi-stage verification pipelines, and show how *Verifloq* can compose the language tier of the verification process.

6.1 Achieved Contributions

Verifloq achieved our initial proposed contributions stated in Section 1.2 in the following ways:

(1) We described our implementation of a strongly typed HDL, and proved it is indeed a type-safe language.
(2) We presented examples supporting a subset of Verilog's language constructs, how they are translated to *Verifloq*'s terms and how they are executed by our proposed semantics.
(3) We presented semantic model and type rules for *Verifloq*, and used the model to prove **Progress** and **Preservation** theorems, guaranteeing verified type-safety.
(4) We presented extensions with Hoare Logic and applications for Program Equivalence.

6.2 Future Work

Developing a semantic model that takes into account all the concurrent aspects of an HDL is a challenging task, and it is currently an open research problem. So far, at this point, we are not aware of any formalization in HOL or any other proof assistants supporting multi-modules. All existing formalizations, including *Verifloq*, are limited to single modules,

synchronous edges. Representing concurrent, mutable shared data across modules is non-trivial. Encoding these aspects in a semantic model using theorem provers or in a proof assistant like Coq is a task that requires further investigation.

Other types of logic, such as Separation Logic (SL) [Reynolds 2002], might be an alternative and fruitful path to explore modules formalization. Traditionally, it has been used for formalizing pointers and memory allocations [Pym et al. 2019]. Its concurrent version (Concurrent Separation Logic, or CSL) has been adopted to formalize shared, mutable data in complex programming languages [Jung et al. 2018].

Although it has not been heavily explored in hardware verification, Separation Logic has been used to verify properties of circuits produced by HLS tools. In their work, [Winterstein et al. 2016] authors SL to reason about properties of the high-level design language. They perform a heap analysis of C/C++ before the synthesis of Verilog code to ensure the HLL design matches is equivalent to the input software specifications that were given to the tool. It would be interesting to create an extension of *Verifloq* combining the ideas of [Winterstein et al. 2016] and Separation Logic to prove correctness of multi-module implementations.

Another aspect to be explored is a better use of dependent types, possibly to encode properties of circuits. The works of [Basin et al. 1991] and [Hanna et al. 1989] illustrate the role of depdent types in formal hardware verification in other proof environments, such as Nuprl. The Coquet library [Braibant 2011] uses dependently typed data-types to reason about the behavior of simple circuits. It would be interesting to extend Verifloq's language constructs with deep embeddings.

A Code Access

Our Coq formalization can be found on https://github.com/galois1/Verifloq. It is currently a private, temporary repository due to approaching and ongoing double-blind submissions. You can request access, or wait for an eventual migration to https://github.com/hannelita/Verifloq .

B Coq tactics and commands

B.1 Verifloq's Coq Definitions

We mention, in Section 3.2, that the Notation command is necessary for converting Verilog's syntax to Coq terms. Listing B.1 specifies an example for the if/else construct. We specify the term translation, the operator precedence and associativity rules.

```
Notation "'if' x y 'else' z" :=
  (tm_if x y z) (in custom implv at level 79,
                    x custom implv at level 89,
                    y custom implv at level 89,
                    z custom implv at level 89,
                    left associativity).
```

Listing B.1. Notation to correlate Verifloq's syntax with Coq's encoding.

In Section 3.2, we mention the need for a substitution operation before stating the step relation of *Verifloq*'s small-step operational semantics. The implementation can be found in Listing B.2.

```
Reserved Notation "'[' x ':=' s ']' t" (in custom implv at level 20, x constr).
Fixpoint substitution (x : string) (s : tm) (t : tm) : tm :=
  match t with
  (* pure STLC *)
  | tm_var y ⇒
      if eqb_string x y then s else t
  | tm_abs y T t1 ⇒
```

```
    if eqb_string x y then t else (tm_abs y T ([ x := s] t1)
| tm_app t1 t2 ⇒
   tm_app ([ x:=s] t1) ([ x:=s] t2)
(* numbers *)
| tm_const _ ⇒
    t
| tm_succ t1 ⇒
    tm_succ ([x := s] t1)
| tm_pred t1 ⇒
    tm_pred ([x := s] t1)
| tm_mult t1 t2 ⇒
    ([ x := s] t1) * ([ x := s] t2>
(* Same for other binary operations, ommiting code here) *
| tm_if t1 t2 t3 ⇒
    tm_if ( ([x:=s] t1) ) ( ([x:=s] t2) ) ( ([x:=s] t3) )
(* sums *)
| tm_inl T2 t1 ⇒
    tm_inl T2 ( [x:=s] t1)
| tm_inr T1 t2 ⇒
    inr T1 ( [x:=s] t2)
| tm_case t0 y1 t1 y2 t2 ⇒
    tm_case ([x:=s] t0)
       y1 (if beq_id x y1 then t1 else ([ x:=s] t1))
       y2 (if beq_id x y2 then t2 else ([ x:=s] t2))
(* lists *)
| tm_nil _ ⇒
```

```
    t
| tm_cons t1 t2 ⇒
    tm_cons ([x:=s] t1) ([ x:=s] t2) >
| tm_lcase t1 t2 y1 y2 t3 ⇒
    tm_lcase ([x:=s] t1) ([ x:=s] t2) y1 y2
      (if beq_id x y1 then
         t3
       else if beq_id x y2 then t3
            else ([ x:=s] t3))
(* unit *)
| tm_unit ⇒ tm_unit
| tm_pair t1 t2 ⇒
    tm_pair ([x:=s] t1) ([ x:=s] t2)
| tm_fst t1 ⇒
    tm_fst ([ x:=s] t1)
| tm_snd t1 ⇒
    tm_snd ([ x:=s] t1)
| tm_rec t ⇒ tm_rec ([x:=s] t)
| tm_emprec _ ⇒
    t
| tm_record ⇒
    tm_record ([x:=s] t1) ([ x:=s] t2)
| tm_lproj t0 y1 t1 y2 t2 ⇒
    tm_case ([x:=s] t0)
       y1 (if beq_id x y1 then t1 else ([ x:=s] t1))
       y2 (if beq_id x y2 then t2 else ([ x:=s] t2))
```

```
  (* assignments *)
  | tm_empasgn _ ⇒
      t
  | tm_asgn t1 t2 y1 y2 t3 ⇒
      tm_asgn ([x:=s] t1) ([x:=s] t2) y1 y2
        (if beq_id x y1 then
           t3
         else if beq_id x y2 then t3
              else ([x:=s] t3))
  | tm_casgn t0 y1 t1 y2 t2 ⇒
      tm_asgn ([x:=s] t0)
         y1 (if beq_id x y1 then t1 else ([x:=s] t1))
         y2 (if beq_id x y2 then t2 else ([x:=s] t2))
  | tm_blocking t1 t2 ⇒
      tm_blocking ([x:=s] t1) ([x:=s] t2)
  | tm_nonblocking t1 t2 ⇒
      tm_blocking ([x:=s] t1) ([x:=s] t2)
  | tm_module t1 t2 t3 t4 ⇒
      tm_module ([x:=s] t1) ([x:=s] t2) ([x:=s] t3) ([x:=s] t4)
  end
where "'[' x ':=' s ']' t" := (substitution x s t) (in custom implv).
```

Listing B.2. Substitution operation.

Also in Section 3.2, we provided the inference rules for *Verifloq*'s small-step operational semantics. The corresponding Coq code can be found in Listing B.3.

```
Inductive step : tm -> tm -> Prop :=
```

```
| Step_AppAbs : forall x T2 t1 v2,
      val v2 ->
      tm_app (tm_abs x T2 t1) v2 ⟶ [x:=v2] t1
| Step_App1 : forall t1 t1' t2,
      t1 ⟶ t1' ->
      tm_app t1 t2 ⟶ tm_app t1' t2
| Step_App2 : forall v1 t2 t2',
      val v1 ->
      t2 ⟶ t2' ->
      tm_app v1 t2 ⟶ tm_app v1 t2'
| Step_Add1 : forall t1 t1' t2,
      t1 ⟶ t1' ->
      tm_add t1 t2 ⟶ tm_add t1' t2
| Step_Add2 : forall v1 t2 t2',
      val v1 ->
      t2 ⟶ t2' ->
      tm_add v1 t2 ⟶ tm_app v1 t2'
| Step_IfTrue : forall t1 t2,
    tm_if true t1 t2 ⟶ t1
| Step_IfFalse : forall t1 t2,
    tm_if false t1 t2 ⟶ t2
| Step_If : forall t1 t1' t2 t3,
    t1 ⟶ t1' ->
    tm_if t1 t2 t3 ⟶ tm_if t1' t2 t3
| Step_Succ : forall t1 t1',
      t1 ⟶ t1' ->
```

```
      tm_succ t1 ⟶ tm_succ t1'
| Step_SuccNat : forall n : nat,
      tm_succ n ⟶ S n
| Step_Pred : forall t1 t1',
      t1 ⟶ t1' ->
      tm_pred t1 ⟶ tm_pred t1'
| Step_PredNat : forall n:nat,
      tm_pred n ⟶ Nat.pred n
| ST_Rec1 : forall t1 t1',
    t1 ⟶ t1' ->
    tm_rec t1 ⟶ tm_rec t1'
| ST_RecAbs : forall xab T1 t2 fn,
    fn = tm_abs xab T1 t2 ->
    tm_rec F ⟶ [xab := (tm_fix fn)] t2
| ST_Pair1 : forall t1 t1' t2,
     t1 ⟶ t1' ->
     (tm_pair t1 t2) ⟶ (tm_pair t1' t2)
| ST_Pair2 : forall v1 t2 t2',
     val v1 ->
     t2 ⟶ t2' ->
     (tm_pair v1 t2) ⟶ (tm_pair v1 t2')
| ST_Fst1 : forall t1 t1',
     t1 ⟶ t1' ->
     (tm_fst t1) ⟶ (tm_fst t1')
| ST_FstVal : forall v1 v2,
     val v1 ->
```

```
      val v2 ->
      (tm_fst (tm_pair v1 v2)) ⟶ v1
| ST_Snd1 : forall t1 t1',
      t1 ⟶ t1' ->
      (tm_snd t1) ⟶ (tm_snd t1')
| ST_SndVal : forall v1 v2,
      val v1 ->
      val v2 ->
      (tm_snd (tm_pair v1 v2)) ⟶ v2
| ST_Inl : forall t1 t1' T,
      t1 ⟶ t1' ->
      (tm_inl T t1) ⟶ (tm_inl T t1')
| ST_Inr : forall t1 t1' T,
      t1 ⟶ t1' ->
      (tm_inr T t1) ⟶ (tm_inr T t1')
| ST_Case : forall t0 t0' x1 t1 x2 t2,
      t0 ⟶ t0' ->
      (tm_case t0 x1 t1 x2 t2) -> (tm_case t0' x1 t1 x2 t2)
| ST_CaseInl : forall v0 x1 t1 x2 t2 T,
      val v0 ->
      (tm_case (tm_inl T v0) x1 t1 x2 t2) ⟶ [x1:=v0] t1
| ST_CaseInr : forall v0 x1 t1 x2 t2 T,
      val v0 ->
      (tm_case (tm_inr T v0) x1 t1 x2 t2) ⟶ [x2 := v0] t2
| ST_Cons1 : forall t1 t1' t2,
     t1 ⟶ t1' ->
```

```
      (tm_cons t1 t2) ⟶ (tm_cons t1' t2)
| ST_Cons2 : forall v1 t2 t2',
      val v1 ->
      t2 ⟶ t2' ->
      (tm_cons v1 t2) ⟶ (tm_cons v1 t2')
| ST_Lcase1 : forall t1 t1'  t2 x1 x2 t3,
      t1 ⟶ t1' ->
      (tm_lcase t1 t2 x1 x2 t3) ⟶ (tm_lcase t1' t2 x1 x2 t3)
| ST_LcaseNil : forall T t2 x1 x2 t3,
      (tm_lcase (tm_nil T) t2 x1 x2 t3) ⟶ t2
| ST_LcaseCons : forall v1 vl t2 x1 x2 t3,
      val v1  ->
      val vl  ->
      (tm_lcase (tm_cons v1 vl) t2 x1 x2 t3) ⟶ ([ x2 := vl ] [(  [x1 :=  v1] t3)] )
| ST_Record : forall t1 t1'  t2,
     t1 ⟶ t1' ->
     (tm_record t1 t2) ⟶ (tm_record t1' t2)
| ST_RecordProj1 : forall t t',
     t ⟶ t' ->
     (tm_lproj t) ⟶ (tm_lproj t')
| ST_RecordProj2 : forall t v,
     val v ->
     (tm_lproj t) ⟶ v
| ST_CaseAsgn : forall t0 t0'  x1 t1 x2 t2,
     t0 ⟶ t0' ->
     (tm_asgn t0 x1 t1 x2 t2) -> (tm_asgn t0' x1 t1 x2 t2)
```

```
| ST_CaseEAsgn : forall t0 x1 t1 x2 t2 v,
    value v ->
    (tm_asgn t0 x1 t1 x2 t2) -> ( [x1 := v] t0)
| ST_Blocking -> forall t1 t1' t2,
    t1 ⟶ t1' ->
    tm_blocking t1 t2 ⟶ tm_blocking t1' t2
| ST_NonBlocking -> forall t1 t1' t2,
    t1 ⟶ t1' ->
    tm_blocking t1 t2 ⟶ tm_blocking t1' t2
| Step_Module : forall t1 t2 t1' l1 l2
    t1 ⟶ t1' ->
    tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 t1' t2
| Step_ModuleMerge : forall t1 t2 t1' l1 l2
    val v=(mstep_merge l1 l2 t1 t2) ->
    t1 ⟶ t1' ->
    tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 v t1'
| Step_ModuleCommit : forall t1 t2 t1' l1 l2 v1
    val v1 ->
    t2 ⟶ val v2=(mstep_commit l1 l2 t1 t2) ->
    tm_module l1 l2 t1 t2 ⟶ tm_module l1 l2 v1 v2
| Step_ModuleEnd : forall t1 t2 t1' l1 l2 v
    val v ->
    tm_module l1 l2 t1 t2 ⟶ v
where "t '⟶' t'" := (step t t').
```

Listing B.3. Step relation for Verifloq's terms.

In Section 3.3, we provided the corresponding typing rules for each of *Verifloq*'s terms. The Coq implementation can be found in Listing B.4.

```
Reserved Notation "Γ '⊢' t '\in' T"
            (at level 10,
             t custom implv, T custom implv at level 0).
Inductive typing : context -> tm -> typ -> Prop :=
  (* pure STLC *)
  | TR_Var : forall Gamma x T1,
      Gamma x = Some T1 ->
      Gamma |- x \in T1
  | TR_Abs : forall Gamma x T1 T2 t1,
    (x |-> T2 ; Gamma) |- t1 \in T1 ->
      Gamma |- \x:T2, t1 \in (T2 -> T1)
  | TR_App : forall T1 T2 Gamma t1 t2,
      Gamma |- t1 \in (T2 -> T1) ->
      Gamma |- t2 \in T2 ->
      Gamma |- t1 t2 \in T1
  (* numbers *)
  | TR_Nat : forall Gamma (n : nat),
      Gamma |- n \in Nat
  | TR_Succ : forall Gamma t,
      Gamma |- t \in Nat ->
      Gamma |- succ t \in Nat
  | TR_Pred : forall Gamma t,
      Gamma |- t \in Nat ->
      Gamma |- pred t \in Nat
```

```
| TR_Mult : forall Gamma t1 t2,
    Gamma |- t1 \in Nat ->
    Gamma |- t2 \in Nat ->
    Gamma |- t1 * t2 \in Nat
| TR_If : forall Gamma t1 t2 t3 T0,
    Gamma |- t1 \in Nat ->
    Gamma |- t2 \in T0 ->
    Gamma |- t3 \in T0 ->
    Gamma |- if t1 then t2 else t3 \in T0
| TR_Pair : forall Gamma t1 t2 T1 T2,
    Gamma |- t1 \in T1 ->
    Gamma |- t2 \in T2 ->
    Gamma |- (tpair t1 t2) \in (TProd T1 T2)
| TR_Fst : forall Gamma t T1 T2,
    Gamma |- t \in (TProd T1 T2) ->
    Gamma |- (tfst t) \in T1
| TR_Snd : forall Gamma t T1 T2,
    Gamma |- t \in (TProd T1 T2) ->
    Gamma |- (tsnd t) \in T2
| TR_Inl : forall Gamma t1 T1 T2,
    Gamma |- t1 \in T1 ->
    Gamma |- (inl T2 t1) \in (T1 + T2)
| TR_Inr : forall Gamma t2 T1 T2,
    Gamma |- t2 \in T2 ->
    Gamma |- (inr T1 t2) \in (T1 + T2)
| TR_Case : forall Gamma t0 x1 T1 t1 x2 T2 t2 T3,
```

```
      Gamma |- t0 \in (T1 + T2) ->
      (x1 |-> T1 ; Gamma) |- t1 \in T3 ->
      (x2 |-> T2 ; Gamma) |- t2 \in T3 ->
      Gamma |- (case t0 of | inl x1 ⇒ t1 | inr x2 ⇒ t2) \in T3
  | TR_Nil : forall Gamma T1,
      Gamma |- (nil T1) \in (List T1)
  | TR_Cons : forall Gamma t1 t2 T1,
      Gamma |- t1 \in T1 ->
      Gamma |- t2 \in (List T1) ->
      Gamma |- (t1 :: t2) \in (List T1)
  | TR_Lcase : forall Gamma t1 T1 t2 x1 x2 t3 T2,
      Gamma |- t1 \in (List T1) ->
      Gamma |- t2 \in T2 ->
      (x1 |-> T1 ; x2 |-> <{{List T1}}> ; Gamma) |- t3 \in T2 ->
      Gamma |- (case t1 of | nil ⇒ t2 | x1 :: x2 ⇒ t3) \in T2
  | TR_Unit : forall Gamma,
      Gamma |- unit \in Unit
  | TR_Rec : forall Gamma t1 T1,
      Gamma |- t1 \in (Ty_Arrow T1 T1) ->
      Gamma |- (tfix t1) \in T1
where "Γ '⊢' t '\in' T" := (typing Γ t T).
```

Listing B.4. Typing relation for Verifloq's terms.

B.2 Coq Proofs

In Chapter 3, we listed relevant proofs, and they are entirely shown in this section.

```
Theorem unique_types : forall Γ e T T',
  Γ ⊢ e \in T ->
  Γ ⊢ e \in T' ->
  T = T'.
Proof with eauto.
  intros. generalize dependent T'.
  induction H; intros.
  - inversion H0; subst. rewrite H3 in H.  injection H as H.
    symmetry in H.  auto.
  - inversion H0; subst. apply IHtyping in H6. subst.
    reflexivity.
  - inversion H1; subst. apply IHtyping1 in H5.
    apply IHtyping2 in H7. inversion H5.  subst. reflexivity.
  - inversion H0; subst. reflexivity.
  - inversion H0; subst. reflexivity.
  - inversion H2; subst. auto.
Qed.
```

Listing B.5. Type Uniqueness.

```
Theorem progress : forall t T,
  · ⊢ t \in T ->
  value t ∨ ∃ t', t ⟶ t'.
Proof with eauto.
  intros t.
  induction t; intros T Ht;  auto.
```

```
  - inversion Ht; subst. inversion H1.
  - inversion Ht; subst.
    remember H2. clear Heqh.
    remember H4. clear Heqh0.
    apply IHt1 in H2.  apply IHt2 in H4.  right.
    destruct H2; destruct H4.
    -- apply (canonical_forms_fun t1 T2 T h) in H.
       inversion H. inversion H1; subst.
       exists (tm_app [x0 := t2] x1 ).   eauto.
    -- inversion H0; subst. exists (tm_app t1 x0).  eauto.
    -- inversion H; subst. exists (tm_app x0 t2).  eauto.
    -- inversion H; subst. exists (tm_app x0 t2).  eauto.
  - inversion Ht; subst. right. remember H3. clear Heqh.
    apply IHt1 in H3.  destruct H3.
    apply (canonical t1 h) in H.  destruct H; subst.
    eauto. eauto.
    inversion H.
    exists (tm_if x0 t2 t3).  eauto.
```

Listing B.6. Progress Theorem.

```
Lemma preservation_subst : forall Gamma x U t v T,
  x ⊢ U ; Γ ⊢ t \in T ->
  · ⊢ v \in U    ->
  Γ ⊢ [x:=v]t \in T.
Proof with eauto.
  intros Gamma x U t v T Ht Hv.
```

```
  remember (x ⊢ U; Gamma) as Gamma'.
  generalize dependent Gamma.
  induction Ht; intros Gamma' G;  simpl; eauto.
  - destruct (eqb_string x x0) eqn:eVt.
    -- subst. apply weakening1.
       apply eqb_str_true in eVt. subst.
       rewrite updt_eq in H.  injection H as H.  subst.
       assumption.
    -- subst. apply T_Var.
       apply eqb_str_false in eVt.
       rewrite update_neq in H. auto.  assumption.
  - destruct (eqb_string x x0) eqn:eVt.
    -- subst. apply eqb_str_true in eVt.
       subst.   rewrite updt_sh in Ht. apply T_Abs.
       assumption.
    -- subst. apply eqb_str_false in eVt.
       apply T_Abs. apply IHHt.
       rewrite updt_map_permute. auto. auto.
Qed.


Theorem preservation : forall t t'  T,
     · ⊢ t \in T ->
     t ⟶ t' ->
     · ⊢ t' \in T.
Proof with eauto.
  intros t t'  T HT.  generalize dependent t'.
```

```
  remember empty as Gamma.
  induction HT;
      intros t' HE; subst;
      try solve [inversion HE; subst; auto].
  - inversion HE; subst...
    -- apply preservation_subst with T2...
       inversion HT1...
Qed.
```

Listing B.7. Preservation Theorem.

```
Lemma substitution_preserves_types : forall Gamma x U t v T,
  x |-> U ; Gamma |- t \in T ->
  empty |- v \in U   ->
  Gamma |- [x:=v]t \in T.
Proof with eauto.
  intros Gamma x U t v T Ht Hv.
  remember (x |-> U; Gamma) as Gamma'.
  generalize dependent Gamma.
  induction Ht; intros Gamma' G; simpl; eauto.
  - destruct (eqb_string x x0) eqn:eVt.
    -- subst. apply weakening_empty. apply eqb_string_true_iff in eVt.
    subst. rewrite update_eq in H. injection H as H. subst. assumption.
    -- subst. apply T_Var. apply eqb_string_false_iff in eVt.
    rewrite update_neq in H. auto. assumption.
  - destruct (eqb_string x x0) eqn:eVt.
    -- subst. apply eqb_string_true_iff in eVt. subst.
```

```
    rewrite update_shadow in Ht. apply T_Abs. assumption.
    -- subst. apply eqb_string_false_iff in eVt. apply T_Abs.
    apply IHHt. rewrite update_permute. auto. auto.
Qed.
```

Listing B.8. Substitution preserves types.

```
Lemma weakening : forall Gamma Gamma' t T,
     inclusion Gamma Gamma' ->
     Gamma  |-  t \in T  ->
     Gamma' |-  t \in T.
Proof.
  intros Gamma Gamma' t T H Ht.
  generalize dependent Gamma'.
  induction Ht; eauto using tac_inclup.
Qed.
```

Listing B.9. Weakening Lemma.

B.3 Interactive theorem proving

Interactive theorem proving consists of "arrangement where the machine and a human user work together interactively to produce a formal proof" [Harrison et al. 2014].

To better illustrate how it works in Coq, Figure B.1 shows the online proof environment *jsCoq*[13], using a proof example from an initial chapter from [de Amorim Chris Casinghino Marco Gaboardi Michael Greenberg Cătălin Hriţcu Vilhelm Sjöberg Brent Yorgey 2020].

[13] https://coq.vercel.app/

On the left side, we see the theorem we want to prove; on the right side, there is the Goals panel, holding the status of the proof.

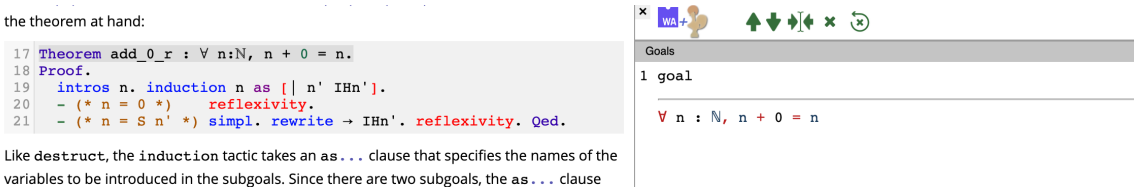

Fig. B.1. An example of an interactive proof environment illustrated by jsCoq.

When we interactively navigate into the proof (Figure B.2), tactic by tactic, we can see the "Goals" panel changing accordingly. Every tactic (if properly used) represents a required step in the proof, and Coq keeps track of what we still need to prove (subgoals as in Figure B.3) or what is the status of the goal.

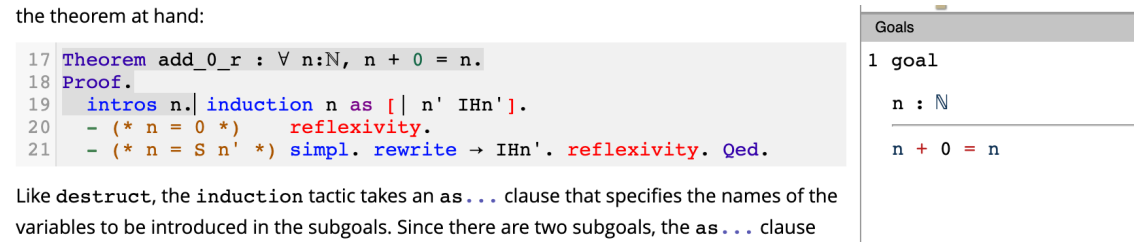

Fig. B.2. Interactive theorem proving: navigating through the tactics.

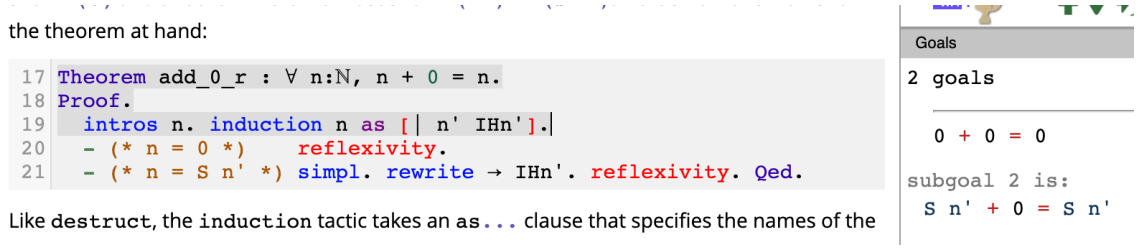

Fig. B.3. Interactive theorem proving: subgoals.

B.4 Relevant Tactics

An extensive list of Coq tactics can be found on the official documentation [14]. This informal, non-official cheat-sheet [15] [16] can be useful for quick reference.

intros: Introduces variables appearing with forall as well as the premises (left-hand side) of implications.

simpl: Simplifies the goal or the hypotheses.

apply: Uses implications to transform goals and hypotheses.

inversion: Reveals other necessary conditions for a hypothesis to be true.

destruct: Generates a subgoal for every constructor of an inductive type.

auto and *eauto*: Solves a greater variety of easy goals.

discriminate: Solves the goal if it is a trivial inequality and solves any goal if the context contains a false equality.

[14] https://coq.inria.fr/refman/proof-engine/tactics.html
[15] https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html
[16] https://www.cs.cornell.edu/courses/cs3110/2018sp/a5/coq-tactics-cheatsheet.html

References

Mark D. Aagaard, Miriam Leeser, and Phillip J. Windley. 1993. Toward a Super Duper Hardware Tactic. In *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 780)*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.). Springer, 399–412. https://doi.org/10.1007/3-540-57826-9_151

Stephen Bailey. 2003. Comparison of vhdl, verilog and systemverilog. *Available for download from www. model. com* (2003), 29.

Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda Calculus with Types*. Cambridge University Press. https://doi.org/10.1017/CBO9781139032636

H. P. Barendregt. 1993. *Lambda Calculi with Types*. Oxford University Press, Inc., USA, 117–309.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. 1997. *The Coq Proof Assistant Reference Manual : Version 6.1*. Research Report RT-0203. INRIA. 214 pages. https://hal.inria.fr/inria-00069968 Projet COQ.

David A. Basin, Geoffrey M. Brown, and Miriam Leeser. 1991. Formally verified synthesis of combinational CMOS circuits. *Integr.* 11, 3 (1991), 235–250. https://doi.org/10.1016/0167-9260(91)90048-P

Lionel Bening and Harry Foster. 2001. *Principles of Verifiable RTL Design* (2nd ed.). Kluwer Academic Publishers, USA.

Janick Bergeron. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, USA.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

Mohammad-Mahdi Bidmeshki, Xiaolong Guo, Raj Gautam Dutta, Yier Jin, and Yiorgos Makris. 2017. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation. *IEEE Transactions on Information Forensics and Security* 12, 10 (2017), 2430–2443. https://doi.org/10.1109/TIFS.2017.2707327

D. Biederman. 1997. An overview on writing a VHDL testbench. In *Proceedings The Twenty-Ninth Southeastern Symposium on System Theory*. 384–388. https://doi.org/10.1109/SSST.1997.581677

David L. Bird and Carlos Urias Munoz. 1983. Automatic Generation of Random Self-Checking Test Cases. *IBM Syst. J.* 22, 3 (1983), 229–245. https://doi.org/10.1147/sj.223.0229

N. Bombieri, F. Fummi, and G. Pravadelli. 2006. On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL. In *Proceedings of the Design Automation & Test in Europe Conference*, Vol. 1. 1–6. https://doi.org/10.1109/DATE.2006.243898

M. Boulé and Z. Zilic. 2005. Incorporating efficient assertion checkers into hardware emulation. In *2005 International Conference on Computer Design*. 221–228. https://doi.org/10.1109/ICCD.2005.66

Marc Boulé and Zeljko Zilic. 2008. *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring* (1 ed.). Springer Publishing Company, Incorporated.

Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 243–257. https://doi.org/10.1145/3385412.3385965

Thomas Braibant. 2011. Coquet: A Coq Library for Verifying Hardware. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7086)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer, 330–345. https://doi.org/10.1007/978-3-642-25379-9_24

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Helge Anderson, Stephen Dean Brown, and Tomasz S. Czajkowski. 2011. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, John Wawrzynek and Katherine Compton (Eds.). ACM, 33–36. https://doi.org/10.1145/1950413.1950423

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

Shivakumar S Chonnad and Needamangalam B Balachander. 2007. *Verilog: Frequently Asked Questions: Language, Applications and Extensions*. Springer Science & Business Media.

Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337

Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic* 5, 2 (1940), 56–68. http://www.jstor.org/stable/2266170

Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. 2009. Model checking: algorithmic verification and debugging. *Commun. ACM* 52, 11 (2009), 74–84. https://doi.org/10.1145/1592761.1592781

Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, Robert Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall. http://dl.acm.org/citation.cfm?id=10510

Thierry Coquand and Gérard P. Huet. 1985. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In *EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 1: Invited Lectures (Lecture Notes in Computer Science, Vol. 203)*, Bruno Buchberger (Ed.). Springer, 151–184. https://doi.org/10.1007/3-540-15983-5_13

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Philippe Coussy and Adam Morawiec. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit* (1st ed.). Springer Publishing Company, Incorporated.

Clifford Cummings. 1999. Correct Methods For Adding Delays To Verilog Behavioral Models. In *International HDL Conference 1999 Proceedings*. 23–29.

H. B. Curry. 1934. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America* 20, 11 (1934), 584–590. http://www.jstor.org/stable/86796

Benjamin C. Pierce Arthur Azevedo de Amorim Chris Casinghino Marco Gaboardi Michael Greenberg Cătălin Hriţcu Vilhelm Sjöberg Brent Yorgey. 2020. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook.

Jack B. Dennis. 2011. *Petri Nets*. Springer US, Boston, MA, 1525–1530. https://doi.org/10.1007/978-0-387-09766-4_134

J. Dimitrov. 2001. Operational semantics for Verilog. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*. 161–168. https://doi.org/10.1109/APSEC.2001.991473

R. Eastham and K. Thirunarayan. 1996. Proof strategies for hardware verification. In *Proceedings of the IEEE 1996 National Aerospace and Electronics Conference NAECON 1996*, Vol. 2. 451–458 vol.2. https://doi.org/10.1109/NAECON.1996.517689

E Engeler. 1984. HP Barendregt. The lambda calculus. Its syntax and semantics. Studies in logic and foundations of mathematics, vol. 103. North-Holland Publishing Company, Amsterdam, New York, and Oxford, 1981, xiv+ 615 pp. *The Journal of Symbolic Logic* 49, 1 (1984), 301–303.

Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2015. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia (LIPIcs, Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

H. Foster. 2001. Applied Boolean equivalence verification and RTL static sign-off. *IEEE Design & Test of Computers* 18, 4 (2001), 6–15. https://doi.org/10.1109/54.936244

Harry Foster, Adam Krolnik, and David Lacey. 2004. *Assertion-based design, Second Edition*. Kluwer.

Steve Golson et al. 1994. State machine design techniques for Verilog and VHDL. *Synopsys Journal of High-Level Design* 9, 1-48 (1994), 12.

Kees G. W. Goossens. 1995. Reasoning about VHDL using operational and observational semantics. In *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95, Frankfurt/Main, Germany, October 2-4, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 987)*, Paolo Camurati and Hans Eveking (Eds.). Springer, 311–327. https://doi.org/10.1007/3-540-60385-9_19

M. Gordon. 1995. The semantic challenge of Verilog HDL. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*. 136–145. https://doi.org/10.1109/LICS.1995.523251

Michael J. C. Gordon. 1988. *HOL: A Proof Generating System for Higher-Order Logic*. Springer US, Boston, MA, 73–128. https://doi.org/10.1007/978-1-4613-2007-4_3

Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. 2016. Scalable SoC trust verification using integrated theorem proving and model checking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 124–129. https://doi.org/10.1109/HST.2016.7495569

F. Keith Hanna, Neil Daeche, and Mark Longley. 1989. Veritas$^+$: A Specification Language Based on Type Theory. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects, Mathematical Science Institute Workshop, Cornall University, Ithaca, New York, USA, July 5-7, 1989, Proceedings (Lecture Notes in Computer Science, Vol. 408)*, Miriam Leeser and Geoffrey Brown (Eds.). Springer, 358–379. https://doi.org/10.1007/0-387-97226-9_37

John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of Interactive Theorem Proving. In *Computational Logic*, Jörg H. Siekmann (Ed.). Handbook of the History of Logic, Vol. 9. Elsevier, 135–214. https://doi.org/10.1016/B978-0-444-51624-4.50004-6

Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485494

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Falk Howar, Bengt Jonsson, and Frits W. Vaandrager. 2019. Combining Black-Box and White-Box Techniques for Learning Register Automata. In *Computing and Software Science - State of the Art and Perspectives*, Bernhard Steffen and Gerhard J. Woeginger (Eds.). Lecture Notes in Computer Science, Vol. 10000. Springer, 563–588. https://doi.org/10.1007/978-3-319-91908-9_26

Std IEEE. 2001. IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001* (2001), 1–792. https://doi.org/10.1109/IEEESTD.2001.93352

Std IEEE. 2005a. IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005* (2005), 1–143. https://doi.org/10.1109/IEEESTD.2005.97780

Std IEEE. 2005b. IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language. *IEEE Std 1800-2005* (2005), 1–648. https://doi.org/10.1109/IEEESTD.2005.97972

Yier Jin, Xiaolong Guo, Raj Gautam Dutta, Mohammad-Mahdi Bidmeshki, and Yiorgos Makris. 2017. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part I: Framework Fundamentals. *IEEE Transactions on Information Forensics and Security* 12, 10 (2017), 2416–2429. https://doi.org/10.1109/TIFS.2017.2707323

Jeffrey J. Joyce. 1990. *More Reasons Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware*. Technical Report. CAN.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 247)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, 22–39. https://doi.org/10.1007/BFb0039592

T. Kam and P.A. Subrahmanyam. 1995. Comparing layouts with HDL models: a formal verification technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14, 4 (1995), 503–509. https://doi.org/10.1109/43.372376

Christoph Kern and Mark R. Greenstreet. 1999. Formal verification in hardware design: a survey. *ACM Trans. Design Autom. Electr. Syst.* 4, 2 (1999), 123–193. https://doi.org/10.1145/307988.307989

Carlos Delgado Kloos and Peter T. Breuer. 1995. *Formal Semantics for VHDL*. Kluwer Academic Publishers, USA.

Alfred Koelbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. 2009. Solver technology for system-level to RTL equivalence checking. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. 196–201. https://doi.org/10.1109/DATE.2009.5090657

Thomas Kropf. 1999. *Introduction to Formal Hardware Verification*. Springer. https://doi.org/10.1007/978-3-662-03809-3

R. Kumar, T. Kropf, and K. Schneider. 1991. First Steps Towards Automating Hardware Proofs In HOL. In *1991 International Workshop on the HOL Theorem Proving System and Its Applications*. 190–193. https://doi.org/10.1109/HOL.1991.596286

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Ralph Loader. 1998. *Notes on simply typed lambda calculus*. University of Edinburgh.

Andreas Lööw. 2021. Lutsig: a verified Verilog compiler for verified circuit development. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 46–60. https://doi.org/10.1145/3437992.3439916

Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for verilog development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, Stefania Gnesi, Nico Plat, Nancy A. Day, and Matteo Rossi (Eds.). IEEE / ACM, 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

Anmol Mathur, Masahiro Fujita, Edmund M. Clarke, and Pascal Urard. 2009. Functional Equivalence Verification Tools in High-Level Synthesis Flows. *IEEE Des. Test Comput.* 26, 4 (2009), 88–95. https://doi.org/10.1109/MDT.2009.79

Anthony McIsaac. 1993. A Formalization of Abstraction in LAMBDA. In *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 780)*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.). Springer, 227–238. https://doi.org/10.1007/3-540-57826-9_138

Patrick Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. 2010. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 179–188. https://doi.org/10.1109/MEMCOD.2010.5558634

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Katell Morin-Allory and Dominique Borrione. 2006. Proven correct monitors from PSL specifications. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, Georges G. E. Gielen (Ed.). European Design and Automation Association, Leuven, Belgium, 1246–1251. https://doi.org/10.1109/DATE.2006.244079

Katell Morin-Allory, Marc Boule, Dominique Borrione, and Zeljko Zilic. 2008. Proving and disproving assertion rewrite rules with automated theorem provers. In *2008 IEEE International High Level Design Validation and Test Workshop*. 56–63. https://doi.org/10.1109/HLDVT.2008.4695875

Peter D. Mosses. 1999. Foundations of Modular SOS. In *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1672)*, Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki (Eds.). Springer, 70–80. https://doi.org/10.1007/3-540-48340-3_7

Peter D. Mosses. 2001. The Varieties of Programming Language Semantics And Their Uses. In *Perspectives of System Informatics*, Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–190.

Rishiyur S. Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE Computer Society, 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

Lawrence C. Paulson. 1994. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*. Lecture Notes in Computer Science, Vol. 828. Springer. https://doi.org/10.1007/BFb0030541

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, and Michael Greenberg. 2021. *Programming Language Foundations*. Software Foundations, Vol. 2. Electronic textbook.

Andrew M Pitts. 1997. Operationally-based theories of program equivalence. *Semantics and Logics of Computation* 14 (1997), 241.

Andrew M. Pitts. 2000. Operational Semantics and Program Equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (Lecture Notes in Computer Science, Vol. 2395)*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer, 378–412. https://doi.org/10.1007/3-540-45699-6_8

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.* 60-61 (2004), 17–139.

Damien Pous and Davide Sangiorgi. 2019. Bisimulation and Coinduction Enhancements: A Historical Perspective. *Formal Aspects Comput.* 31, 6 (2019), 733–749. https://doi.org/10.1007/s00165-019-00497-w

Vaughan R. Pratt. 1976. SEMANTICAL CONSIDERATIONS ON FLOYD-HOARE LOGIC. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 109–121. https://doi.org/10.1109/SFCS.1976.27

Nitin Pundir, Farimah Farahmandi, and Mark Tehranipoor. 2021. Secure High-Level Synthesis: Challenges and Solutions. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 164–171. https://doi.org/10.1109/ISQED51717.2021.9424365

David Pym, Jonathan M Spring, and Peter O'Hearn. 2019. Why separation logic works. *Philosophy & Technology* 32, 3 (2019), 483–516.

Jaan Raik, Hideo Fujiwara, Raimund Ubar, and Anna Krivenko. 2008. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In *2008 17th Asian Test Symposium*. 21–26. https://doi.org/10.1109/ATS.2008.22

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

Thomas N. Reynolds, Adam M. Procter, William L. Harrison, and Gerard Allwein. 2019. The Mechanized Marriage of Effects and Monads with Applications to High-assurance Hardware. *ACM Trans. Embed. Comput. Syst.* 18, 1 (2019), 6:1–6:26. https://doi.org/10.1145/3274282

Vivek Sagdeo. 2007. *The complete Verilog book*. Springer Science & Business Media.

David A. Schmidt. 1996. Programming Language Semantics. *ACM Comput. Surv.* 28, 1 (1996), 265–267. https://doi.org/10.1145/234313.234419

Dana S. Scott. 1972. Mathematical concepts in programming language semantics. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972 (AFIPS Conference Proceedings, Vol. 40)*. AFIPS, 225–234. https://doi.org/10.1145/1478873.1478903

Natarajan Shankar, Sam Owre, John M Rushby, and Dave WJ Stringer-Calvert. 2001. PVS prover guide. *Computer Science Laboratory, SRI International, Menlo Park, CA* 1 (2001), 11–12.

Alex Simpson and Niels F. W. Voorneveld. 2020. Behavioural Equivalence via Modalities for Algebraic Effects. *ACM Trans. Program. Lang. Syst.* 42, 1 (2020), 4:1–4:45. https://doi.org/10.1145/3363518

Sangeetha Sudhakrishnan, Janaki T. Madhavan, E. James Whitehead Jr., and Jose Renau. 2008. Understanding bug fix patterns in verilog. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey (Eds.). ACM, 39–42. https://doi.org/10.1145/1370750.1370761

Yunfeng Tao. 2009. An introduction to assertion-based verification. In *2009 IEEE 8th International Conference on ASIC*. 1318–1323. https://doi.org/10.1109/ASICON.2009.5351246

Vaibbhav Taraate. 2022. *Concept of Concurrency and Verilog Operators*. Springer Singapore, Singapore, 21–43. https://doi.org/10.1007/978-981-16-3199-3_2

Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 408–422. https://doi.org/10.1007/978-3-540-31987-0_28

G. Umbreit. 1992. Providing a VHDL-interface for proof systems. In *Proceedings EURO-DAC '92: European Design Automation Conference*. 698–703. https://doi.org/10.1109/EURDAC.1992.246187

S. Vasudevan, J.A. Abraham, V. Viswanath, and Jiajin Tu. 2006. Automatic decomposition for sequential equivalence checking of system level and RTL descriptions. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.* 71–80. https://doi.org/10.1109/MEMCOD.2006.1695903

Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 1–14. https://doi.org/10.1145/143165.143169

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

Felix J. Winterstein, Samuel R. Bayliss, and George A. Constantinides. 2016. Separation Logic for High-Level Synthesis. *ACM Trans. Reconfigurable Technol. Syst.* 9, 2 (2016), 10:1–10:23. https://doi.org/10.1145/2836169

Li Yongjian and He Jifeng. 2003. Towards a theory of bisimulation for a fragment of Verilog. In *Proceedings International Parallel and Distributed Processing Symposium*. 8 pp.–. https://doi.org/10.1109/IPDPS.2003.1213435