FORMAL VERIFICATION OF STRUCTURED TEXT PLC CODE USING COQ

AN HONORS THESIS

SUBMITTED ON THE FIFTH DAY OF MAY, 2023

TO THE DEPARTMENTS OF MATHEMATICS AND COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

OF THE HONORS PROGRAM

OF NEWCOMB-TULANE COLLEGE

TULANE UNIVERSITY

FOR THE DEGREE OF

BACHELOR OF SCIENCE

WITH HONORS IN MATHEMATICS AND COMPUTER SCIENCE

BY

Bennett Kahn

APPROVED:

Marie Dahleh, Ph.D.
Director of Thesis

Nicholas Mattei, Ph.D.
Second Reader

Meera Sridhar, Ph.D.
Third Reader
Bennett Kahn. Formal Verification of Structured Text PLC Code Using Coq
(Professors Marie Dahleh and Nicholas Mattei, Mathematics and Computer Science)

Industrial Control Systems (ICS) are the electronic control mechanisms behind factories, power grids, telecommunication infrastructure, nuclear plants, and countless other systems. The main control units of ICS are Programmable Logic Controllers (PLCs). Due to their running of an operating system and their connection to the internet, PLCs are perhaps the largest attack vector of ICS. As safety-critical systems, the security and reliability of ICS—and thus PLCs—are paramount; a compromised system can present disastrous, or even deadly, consequences.

While the research concerning the application of formal verification to PLCs is relatively abundant, the actual adoption of this technique is quite limited. One hindrance to the adoption is the lack of rigorously defined languages for programming PLCs, making the development of formal models difficult. As such, much of the relevant literature focuses on developing these formal models. Out of these works, a small fraction develop formal models for the Structured Text (ST) programming language, one of the five languages recognized by the IEC 61131-3 for programming PLCs, and the most common language for large-scale PLC projects. In this work, we present a formal model of a subset of ST. This model is implemented using Coq, a popular proof assistant framework. To showcase and validate our model, we write a number of ST programs, translate those programs into our model, write specifications defining the intended behavior of each program, and use Coq’s proof solving capabilities to prove that the programs adhere to their specifications.
Acknowledgments

I would like to sincerely thank each of my advisors. Not only did they help me refine my thesis topic, but all three of them have been immensely inspirational and supportive during my college experience. Dr. Mattei’s Artificial Intelligence class my sophomore year really opened my eyes to the beauty of programming and helped me develop my love for computer science. His enthusiasm and knowledge are both seemingly endless, and in every class I have taken with him, I have left inspired by the subject matter. Dr. Dahleh’s comparable enthusiasm for mathematics has heightened my own enthusiasm for the subject. I would like to thank Dr. Dahleh for the many letters of recommendation she has written for me for various research programs over the years. Additionally, I thank Dr. Dahleh for so willingly taking on the role of my math advisor on this thesis, even though I was initially not certain how my work would incorporate math and, not coming from a programming background, she was admittedly not familiar with much of the subject matter. Last but not least, I would like to thank Dr. Sridhar for being a fantastic research mentor to me these last few years, providing me with opportunities to conduct research on topics of great interest to me and to co-author my first published paper. Without her, I would not have even known about the existence of formal methods, and this thesis would have never existed.
# Contents

1 Introduction 1

1.1 Contributions 2

2 Background 2

2.1 PLC & Structured Text 3

2.2 Formal Verification 6

2.3 Coq 8

3 Formalizing ST in Coq 17

3.1 Formalizing the Syntax of ST in Coq 17

3.2 Formalizing the Semantics of ST in Coq 22

4 Formally Verifying ST Programs 27

4.1 Example 1: A Simple Kinetic Energy Program 27

4.2 Example 2: A Simple Summation Program 35

4.3 Example 3: A Factorial Program 38

5 Conclusion 41

6 Future Work 41
1 Introduction

*Industrial Control Systems* (ICS) power factories, farms, oil refineries, telecommunication infrastructure, and much more [41]. Given society’s massive reliance on ICS, the reliability, security, and safety of these systems are imperative. Unfortunately, these systems have recently been the target of several high-profile attacks, such as Black Energy [4], Havex [1], TRITON [19], Stuxnet [30], and the German Steel Mill [35]. Perhaps the largest attack vector of ICS are the many *Programmable Logic Controllers* (PLCs), which are micro-controller based control systems that take input from a number of sensors and output signals to control actuators, e.g., motors and relays [3].

The security community has researched and proposed a number of defense methods for PLCs [43]. Among the most promising methods for defending against attacks targeting PLCs are *formal methods* [20], which, by mathematically proving that a program adheres to formal specifications, aims to verify that a program does or does not perform its expected behavior. For this reason, formal methods are generally recommended for *safety-critical* systems like ICS. Unfortunately, the PLC community has been slow to adopt formal methods at scale, largely due to the difficulties involved in such an application [16]. One hindrance to a large-scale adoption is the fact that formal methods require both a formal model of the PLC program in question, in addition to the formally defined properties the program should satisfy [18]. These models and properties must be created manually, which calls for collaboration between a formal methods expert and a PLC engineer–a workflow that can lead to incorrect models or properties if the PLC program is not understood properly. A second hindrance to the adoption is that many of the programming languages used to program PLCs lack rigorously-defined semantics, which makes developing models for these languages difficult [18]. Accordingly, many works focus entirely on developing models for PLC languages [26, 37, 34, 27, 28, 7, 9, 46, 10, 6, 8, 17]. Out of these works, however, very few present models for the Structured Text (ST) programming language, one of the five International Electrotechnical Commission (IEC) 61131-3 standard [29] languages used to
program PLCs. ST is a text-based language that is supported by some of the largest PLC vendors, such as Omron, Beckhoff, and Siemens, and is able to perform more complex tasks than the other PLC programming languages; accordingly, ST is the most popular language for large-scale PLC projects, such as ICS [26].

1.1 Contributions

In this work, we present a model that defines the syntax and semantics of a subset of the Structured Text (ST) programming language. We create this model using Coq [13], a popular proof assistant technology. We manually convert a number of ST programs into this model and write specifications that these programs should adhere to. Then, we use Coq to prove that the programs (their representation in our model, specifically) meet their specifications (i.e., they behave correctly). Of the formal ST models presented in the literature, Huang et al.’s KST model [26] is perhaps the most complete. KST is implemented in the K Framework [45], a framework for modeling programming languages. We are aware of no attempts in the literature to model ST using Coq. Given the relative novelty of applying formal methods to PLCs, it is worth exploring the capabilities of all relevant frameworks in creating formal models for PLC programming languages. It is very possible that KST may be better suited than a similarly complete ST model in Coq for realistic PLC applications. We do not attempt to argue either side in this work. Rather, the goal of this work is to show that Coq can be used to perform formal verification on ST programs.

2 Background

In this section we provide a brief background of formal verification, PLCs & the ST language, and Coq.
2.1 PLC & Structured Text

PLCs are micro-controller based control systems [3]. Figure 1 depicts a typical PLC and its main components. A PLC takes input from a number of sensors and outputs signals to control actuators, e.g., motors and relays. The main component of a PLC is the Central Processing Unit (CPU), which runs all processes required by the PLC. PLCs operate by running four steps in a continuous loop [3]:

1. the *Input Scan* phase discerns the state of all connected input devices.

2. in the *Program Scan* phase, the PLC program runs to determine output signals.

3. the *Output Scan* phase sends energy signals to connected actuators.

4. in the *Housekeeping* phase, the PLC communicates with exterior infrastructure and runs diagnostics.

As defined by the IEC 61131-3 standard, there are five common languages for programming PLCs: Ladder Diagram (LD), Function Block Diagram (FBD), Instruction List (IL),
Sequential Function Chart (SFC), and Structured Text \[3\]. In this work we are concerned with Structured Text programs.

**Structured Text (ST)** is an abstract, textual language for programming PLCs. ST’s syntax is similar to Pascal, the programming language that ST is based on. As a text-based, typed language, ST allows for the development of complex algorithms. Compared to the other PLC languages that the IEC 61131-3 defines, ST is capable of handling the most advanced tasks \[26\]. Because of this ST is generally the language of choice for large PLC projects. The IEC 61131-3 introduces three *program organization units* (POUs), which are the building blocks of PLC software; they are \[29\]:

1. **Function** - lacks static variables and can be called by other POUs; always returns the same value when given the same input.

2. **Function Block** - contains static variables and has a unique id for each instance; can return different value when given the same input.

3. **Program** - the main loop of the PLC; has access to memory addresses of physical I/O devices; only the PLC itself can call a Program.

An ST program is a collection of POUs \[29\] (at least one must be a program POU). Figure 2 shows examples of each of the three ST POUs. In 2b a function POU named Mul takes two integers, a and b, as input and returns their product. In 2c a function block POU named Counter defines one integer variable called count and increments count. 2a depicts a program POU that invokes both the Mul and Counter POUs.

**PLC Security and Reliability:** PLCs are the computer of choice in many Industrial Control Systems (ICS). As a result, PLCs control crucial systems, such as water treatment facilities, nuclear power plants, and power grids \[41\]. Such systems are crucial to the successful operation of a country, and the consequences of them being inoperative can be disastrous,
or even deadly. While PLCs are generally considered secure and reliable, recent major attacks have shown the need for further advances in the security and reliability of such systems. Perhaps the most well-known of these attacks is the attack in 2010 against Iranian nuclear power plants, which was carried out by the Stuxnet malware [30]. Rumored to be created by both the US and Israeli governments, Stuxnet is an extremely powerful computer worm that exploits zero-day vulnerabilities in Microsoft Windows operating system before propagating its way into Siemens Step7 software that is used to control many PLCs [36, 30]. With a PLC compromised, Stuxnet is able to disrupt the electromechanical processes controlled by that PLC [30]. After the proven success of Stuxnet, several other high profile attacks against PLCs began occurring, such as Black Energy, Havex, Flame [44], Wiper [47], and TRITON. In fact, according to a 2015 study by Kaspersky Lab [31], after the Stuxnet attack against Iran the average number of ICS/PLC attacks increased by about 5% each year. The same report identified 91% of these as medium or critical risk and noted that 15% of the vulnerabilities that these attacks targeted remained unpatched, partially patched, or unable to be
fixed—meaning that attackers could still potentially exploit many of them.

2.2 Formal Verification

In the 1970s, to promote proof-based approaches to verifying programs, the famous computer scientist Edward Wybe Dijkstra once stated, “Testing shows the presence, not the absence, of bugs” [14]. As defined by Edwards et al., formal verification “is the process of mathematically checking that the behavior of a system, described using a formal model, satisfies a given property, also described using a formal model” [20]. In simpler terms, formal verification, or formal methods, attempts to verify that a program is correct; i.e. that it performs its intended behavior, which we model as a collection of properties. These properties, called formal specifications, generally fall into one of two classes:

1. Safety properties state that a system does not reach an undesired state, regardless of given inputs [20].

2. Liveness properties assure the frequency and duration that a system shall arrive in some state [20].

The Origins of Formal Methods: Much of the theory behind formal methods dates back to the 1960s (it would be a while before the theory saw actual applications). In 1969 Tony Hoare developed a type of logic—now known as Hoare Logic— to reason about programs [24]. A visionary in the field of program verification, he asserted, “When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.” [24] Hoare was proposing complete program verification; for decades, his idea was met with much scrutiny [39]. Perhaps most of this scrutiny was related to the high cost associated with complete program verification. In recent years, however, formal methods have grown in popularity [43].
The Abundance and Importance of Formal Methods: With (a) the rapid increase of computing resources, (b) the adoption of cloud-computing, and (c) the growing dependence on computer systems, the computing community has begun to reconsider, and slowly adopt, software verification in the last couple decades, especially in safety-critical industries, like aerospace, medicine, automotive, and defense \cite{33, 42, 11}. (a) has decreased the price of formal methods, (b) means that more and more services are relying on the same computer systems, and (c) engenders a need for provably correct code and security properties. As a result, the research community has proposed several approaches to formal verification, and various sectors have adopted some of these approaches \cite{20, 43}. We are concerned with theorem proving methods only, which aim to provide formal, mathematical proofs that a system satisfies its properties \cite{20}. Software verification has clear cybersecurity implications, as software bugs cause cyber vulnerabilities; software verification attempts to produce rigorously-defined, bug-free, and correct code. Producing correct code, however, extends even beyond the realm of cybersecurity. In addition to facilitating vulnerabilities, software bugs can produce unintended behavior in a program. Unintended behavior is at best unwanted, and at worst deadly. Consider a program in a micro-controller of an airplane; we can imagine what would happen if this program behaved erratically.

In the last few years, several major companies and government organizations have been investing in formal methods, highlighting the recent approach to further expand the adoption of formal methods. The National Science Foundation (NSF), for instance, recently began their Formal Methods in the Field (FMitF) program, a research initiative aimed to advance the body of knowledge concerning formal methods, with an estimated funding amount of $10 million \cite{21}. Additionally, the Defense Advanced Research Projects Agency (DARPA) has recently initiated their High-Assurance Cyber Military Systems (HACMS), aimed to “create technology for the construction of high-assurance cyber-physical systems, where high assurance is defined to mean functionally correct and satisfying appropriate safety and security properties.” \cite{15} Major companies have also been pouring money into formal
methods research. AWS, for example, has been investing in the use of SMT solvers for code verification and cybersecurity [40]. With the current government and private-sector research initiatives towards formal methods, the importance of work dedicated to this field is readily evident.

2.3 Coq

Coq [13] is a proof assistant technology that aids humans in the construction of machine-checked proofs. Coq offers a **vernacular** command language, which allows users to define functions; state theorems and specifications; and develop proofs. Through a **tactic** language, users can interactively develop proofs for these theorems and specifications.

In both pure mathematics and software development, there are several well-known applications that have employed Coq. In 2008 Georges Gonthier developed a machine-checked proof of the four-color-theorem, a mathematical problem with too many constraints to be proven by a human [22]. CompCert [32], meanwhile, is a C Compiler back-end fully verified with Coq. Following are some examples showcasing Coq.

**Example 1: A simple Coq program to show that the addition of natural numbers is commutative:** It is helpful to liken the Coq code and proof we will present to standard mathematics. In math we can state the commutative property of addition for natural numbers as the following theorem:

**Theorem 2.1.** Addition of the Natural Numbers is Commutative. That is, for all natural numbers \(a, b\), \(a + b = b + a\). More formally, \((\forall a, b \in \mathbb{N})(a + b = b + a)\).

In Coq we can assert an identical theorem as follows:

```
1 Theorem add_comm : forall (n m : nat),
2     n + m = m + n.
```

Listing 1: Natural Numbers Commutative on Addition Theorem
Unlike many languages, where an entire file is run at once time, Coq has functionality to run only portions of the code; specifically, stepping one instruction down continues execution until the next . character. After stating a Theorem in Coq, we then type the Proof. keyword, at which point Coq enters proof mode. Accompanying proof mode is a proof view displaying our current subgoals, which are the statements we are trying to prove. After Proof. our proof state is as follows:

\[(1/1)\]
\[\forall n \ m : \text{nat}, \ n + m = m + n\]

In proof mode, we enter a series of commands, known as tactics, to sequentially simplify the subgoals. Anytime there are variables in our subgoal, we need to introduce them. We can do this with the intros tactic in Coq. Here, we have the variables \( m \) and \( n \), so we must introduce them. After running intros n m., our proof state is as follows:

\[\text{n, m: nat}\]
\[\text{--------------------------}\]
\[(1/1)\]
\[\text{n + m = m + n}\]

Consider everything above the dashed line as variables in the statement we are trying to prove, which is below the line. We are going to perform a proof by induction to prove this theorem. We issue the tactic induction n as \([|k \IHk]\). which tells Coq to perform induction on the variable \( n \), using \( k \) as a variable to represent a specific instance of \( n \), and \( \IHk \) as a variable name for the inductive hypothesis. Our proof state:

\[\text{m: nat}\]
\[\text{--------------------------}\]
\[(1/2)\]
\[0 + m = m + 0\]
\[(2/2)\]
\[S k + m = m + S k\]
We now have two subgoals, one for the base case (subgoal 1/2) where \( n \) equals 0 and one for the inductive case (subgoal 2/2) where \( n = k \). These two cases are the basis of mathematical induction. In Coq we work on one subgoal at a time. We can hone in on the first subgoal with a \(-\), after which executing tactics will operate on that subgoal only. Looking at the first subgoal, we see that we have \( 0 + m = m + 0 \). In mathematics we would simplify this expression to arrive at \( m = m \). Coq offers a \texttt{simpl} tactic for precisely that. \texttt{simpl.} gives us:

\[
\begin{align*}
\text{m: nat} \\
\text{-----------------------------} \\
\text{1/1} \\
\text{m = m + 0}
\end{align*}
\]

Notice that Coq has only simplified the left hand side. This is due to way that Coq represents the Natural numbers (\( \mathbb{N} = \{1, 2, 3, 4, \ldots\} \)); in Coq \texttt{nats}, one of the many built in data types of the language, represent the Natural numbers. Because of the way that \texttt{nat} implements addition (+), the fact that \( 0 + m = m \) is trivial, but that \( m + 0 = m \) is not; thus, the \texttt{simpl} tactic cannot simplify it. However, one of the many benefits of Coq are the extensive libraries containing many well-known mathematical theorems and statements, which can be called upon when needed in a proof. For example, the \texttt{nat} data type of Coq has the theorem \texttt{add_0_r}, which can simplify \( m + 0 = m \). Specifically, \texttt{Nat.add_0_r} is defined as follows:

\[
\text{Theorem add_0_r : forall n, n + 0 == n.}
\]

In our subgoal, we have an expression of the form \( n + 0 \) (i.e. \( m + 0 \)), on which we want to apply this theorem; i.e. we want to \texttt{rewrite}, the expression we have, using a theorem. Coq offers the \texttt{rewrite} tactic for this. We want to rewrite an expression matching the left side of the theorem to make it match the right side (a left to right rewrite), so we use \texttt{->} with the tactic. Applying \texttt{rewrite -> Nat.add_0_r}, our proof state is:
We now want to apply the reflexivity property of math, which states that a number always equals itself. Coq offers the reflexivity tactic. Issuing it, our proof state:

There are unfocused goals.

The above message indicates that we have officially proven our current subgoal and have others left to prove. We hone in on the second subgoal with -. Our proof state:

Note the terms above the dashed line. We have naturals k and m in our scope. We also have a variable called IHk which is our inductive hypothesis. As seen, our subgoal has the terms S k, meaning “successor of k”. Coq defines natural numbers according to the Peano axioms, a set of rules intended to formalize natural numbers; one of the axioms states that each natural number n has a successor equal to n + 1 and denoted, S(n) [25]. Thus, S k above means k + 1. We proceed in our proof with simpl.\(^1\)

\(^1\)It is often a good idea to try the simpl tactic once variables have been introduced. It can often massage the statement to something more familiar.
simpl reduces complex terms to simpler terms. Here Coq reduces $S(k + m)$ to one natural number instance, $S(k + m)$, or “the successor of some natural $k + m$”. We note that we have a term equivalent to one side of the inductive hypothesis, indicating it might be a good time to apply it. Like we did for the add_0_r theorem, we can rewrite using IHk. Running rewrite IHk, our proof state is:

```
 k, m: nat
 IHk: k + m = m + k
```

As seen above, Coq has applied IHk to rewrite $S(k + m)$ as $S(m + k)$. The statement we are trying to prove now is its own builtin provable theorem in Coq, defined as plus_n_Sm. Thus, we can just perform a rewrite based on it: rewrite -> plus_n_Sm. gives:

```
 k, m: nat
 IHk: k + m = m + k
```

Each side of the subgoal is the same, so a simple reflexivity. completes our proof, giving No more subgoals in the proof view.

```
Proof.
 intros n m. induction n as [|k IHk].
  - simpl. rewrite -> Nat.add_0_r. reflexivity.
  - simpl. rewrite IHk. rewrite -> plus_n_Sm. reflexivity.
Qed.
```

Listing 2: Final proof of Commutativity of Natural Numbers theorem

Listing 2 shows the complete proof for the add_comm theorem shown in Listing 1. Note

²https://coq.inria.fr/library/Coq.Init.Peano.html#plus_Sn_m
the completion of our proof with the Qed keyword, which removes Coq from proof mode.

Example 2: Proving an implementation of the factorial function in Coq In this example, we provide a recursive implementation of the factorial function, \( \text{factorial}(n) = n \times (n-1) \times (n-2) \times ... \times 1 \), along with an inductive formal specification of the function. We then provide a proof to show that the recursive implementation adheres to the formal specification. While the factorial function is inherently a mathematical concept, the purpose of this example is to showcase how we can write a function, or a program, and use formal methods to prove the implementation is correct. We attribute this example to Cornell’s CS 3110 course notes.

Firstly, we define the recursive factorial function, using Coq’s Fixpoint keyword for recursion:

```
Fixpoint fact (n : nat) :=
  match n with
  | O => 1
  | S k => n * (fact k)
end.
```

Listing 3: Recursive implementation of the factorial function in Coq

The function in Listing 3 has name fact and takes in one parameter \( n \) of type nat. It then recursively performs pattern matching on \( n \) to compute the answer.

Listing 4 shows our formal specification of the factorial function. This is our hard and fast, trusted rule for the factorial function. We use the Inductive keyword to inductively define a proposition factorial_of, which takes in two nats and is a proposition (i.e. Prop).

```
Inductive factorial_of : nat -> nat -> Prop :=
  | factorial_of_zero : factorial_of 0 1
  | factorial_of_succ : forall (a b : nat), factorial_of a b -> factorial_of (S a) ((S a) * b).
```

Listing 4: Specification of factorial function in Coq

\(^3\)https://www.cs.cornell.edu/courses/cs3110/2019sp/lec/26-coq-verification/notes.html
Our proposition has two cases:

1. \texttt{factorial\_of\_zero} on line 2, which states that the factorial of 0 is 1.

2. \texttt{factorial\_of\_succ} on lines 3 and 4, which states that, for all \( a, b \) of type \texttt{nat}, the factorial of \( a \) being equal to \( b \) implies that the factorial of \( S\ a \) equals \( (S\ a) \times b \). In other words, if the factorial of \( a \) is \( b \), the factorial of \( a + 1 \) is \((a + 1) \times b\). This follows directly from the definition of factorial.

\begin{verbatim}
1 Theorem fact_correct : forall (n : nat),
2 factorial_of n (fact n).
\end{verbatim}

Listing 5: Theorem stating that our \texttt{fact} function adheres to our formally specified definition of factorial, \texttt{factorial\_of}

Now that we have our factorial function and its accompanying formal specification, we define a theorem, \texttt{fact\_correct}, which asserts that the function value matches the specification value for all \texttt{nat}s. Listing 5 shows the \texttt{fact\_correct} theorem.

We are now ready to perform the proof. We enter proof mode with \texttt{Proof}:

1/1
\texttt{forall n : nat, factorial_of n (fact n)}

We note the occurrence of a variable, \( n \). In Coq we must introduce any variables in our statement to prove; here, we do so with \texttt{intros n}:

n: nat

================================
1/1
\texttt{factorial_of n (fact n)}

Like in our previous example, we proceed with a proof by induction, issuing the tactic \texttt{instance induction n as [\mid k IHk\mid]}:

1/2
factorial_of 0 (fact 0)
2/2
factorial_of (S k) (fact (S k))

Note our subgoals for the base and inductive cases, respectively. We hone in on the base case subgoal (1/2) with -..:

1/1
factorial_of 0 (fact 0)

We note the occurrence of fact 0, which is an expression that can be directly computed by Coq, so we execute simpl.:

1/1
factorial_of 0 1
fact 0 simplifies to 1. We now have an expression directly resembling our formally specified proposition factorial_of_0. We apply this proposition with the apply tactic: apply factorial_of_zero.: There are unfocused goals.

This message indicates the completion of the proof for the base case, so we hone in on the inductive case subgoal with -..

k: nat
IHk: factorial_of k (fact k)
==================================
1/1
factorial_of (S k) (fact (S k))
simpl. gives:

k: nat
IHk: factorial_of k (fact k)
==================================
factorial_of (S k)

(fact k + k * fact k)

Coq has used the pattern matching rules in fact to simplify fact (S k). We see that our expression resembles the formally specified factorial_of_succ, so we apply it: apply factorial_of_succ.:

k: nat
IHk: factorial_of k (fact k)

Our expression now exactly resemble our inductive hypothesis, IHk. In other words, the subgoal is already in the proof context. The assumption tactic completes goals in such cases. assumption gives:

No more subgoals.

We have now successfully completed our proof. In doing so, we have proven that our original specification of the factorial function is correct. Of course, this relies on the correct implementation of our specification of the factorial numbers. In this case, we write a specification for a very simple function. For much larger, complex programs, however, the formal specifications are much shorter, simpler, and direct than the actual program. In such cases, it is easier to trust the correct implementation of the specification as opposed to the actual program. Another point to note is that, for both simple functions (i.e. fact) and complex programs, the very fact of proving that the implementation meets the specification acts as an added layer of confidence that the program performs its intended behavior.

Listing 6 shows the entirety of the proof of the fact_correct Theorem.
Formalizing ST in Coq

While Coq is a powerful tool, it only understands its own syntax and semantics. In order to formally verify properties about other languages (i.e. use Coq as a language verification framework), Coq must be able to reason about that language. To do this, one must create (or use) a formal model $M$ for that language in Coq. Specifically, $M$ must unambiguously define the target language accurately enough so that formal reasoning is possible, yet abstractly enough so that its loyalty to the target language is verifiable.

Thus, our first step in formally verifying ST programs in Coq is to develop a formal model $M$ of ST’s syntax and semantics. As a starting point for our model, we use Coq code provided by Runtime Verification [2], which discusses general principles for using Coq to verify other programming languages; in this blog post, the authors present a model of a fictional, C-like language called IMP. We alter this model to account for the differences in syntax and semantics between IMP and ST. In the following two subsections, we discuss $M$’s syntax and semantics, respectively.

### 3.1 Formalizing the Syntax of ST in Coq

Like any programming language, ST consists of expressions; an expression is a collection of variables, data types, and operators that yields a value when evaluated [29]. The two most common types of expressions in ST are arithmetic expressions, which operate on numeric
Listing 7: Declaration of new type to represent ST arithmetic expressions in Coq

data types and boolean expressions, which operate on boolean data. We model the syntax of both arithmetic and boolean expressions by using Coq’s Inductive keyword to create new native data types for both arithmetic and boolean expressions.

Listing 7 shows the declaration of a new data type, ArithExpression, whose members are var, sint, int, etc. With this new type, we can use the member constructors to create instances of ArithExpression. We can break the members of ArithExpression into three classes: (1) variable (var), (2) numeric data types (sint through ulint), and (3) operators (add through div). var acts as a construct to store variable names; var’s constructor (var : string -> ArithExpression) specifies that var takes one string argument. Numeric data types are objects that represent ST numbers in Coq. int (5), for instance, represents an ST 5. Operators, finally, are objects to represent arithmetic operations on numeric data types. For example, add (int 5) (int 4) has type ArithExpression and represents the addition of 5 and 4.

Similarly, Listing 8 shows the declaration of a second data type, BooleanExpr. Again,
we can break down the members of BooleanExpr into three categories: (1) boolean data type (bool_cond) which stores ST "True" or "False", (2) logical operators (Lines 4-7) which define common boolean logical operators, and (3) relational operators, which compare two arithmetic expressions in order to return a boolean value.

A ST program is merely a collection of statements. Accordingly, we must next define the syntax of all statements in ST. To do so, we create a new data type Statement, whose members are the syntax for a specific type of statement. Firstly, our model must be able to handle assignments to variables. For example, we should be able to assign a variable "silly_var" the value 4. We create assignment members for both arithmetic and boolean expressions with assign_arith : string -> ArithExpression -> Statement and assign_bool : string -> BooleanExpr -> Statement, respectively. In the following paragraphs, we outline the syntax of other ST statements.

**IF statements** ST defines IF statements as follows:

```plaintext
IF [boolean expression] THEN
   <statement>;
ELSIF [boolean expression] THEN
```
The ELSIF and ELSE portions are optional. An object representing an ST IF statement must have the ability to store a boolean expression, a statement for the body of the IF, and a second statement for the body of the ELSE. We represent IF statements in our model as if_ : BooleanExpr -> Statement -> Statement -> Statement. To model an IF statement that does contain an ELSIF portion, we can simply pass an if_ instance into the final Statement argument of the constructor.

**FOR Loops**  ST defines FOR loops as follows:

```plaintext
FOR count := initial_value TO final_value BY increment DO
    <statement>;
END_FOR;
```

We represent FOR loops in our Coq model as for_ : string -> ArithExpression -> ArithExpression -> Statement -> Statement.

**WHILE Loops**  ST defines WHILE loops as follows:

```plaintext
WHILE [boolean expression] DO
    <statement>;
END_WHILE;
```

We represent WHILE loops in our Coq model as while : BooleanExpr -> Statement -> Statement.

**REPEAT Loops**  ST defines REPEAT loops as follows:

```plaintext
REPEAT
```
We represent REPEAT loops in our Coq model as \texttt{repeat} : \texttt{Statement} \rightarrow \texttt{BooleanExpr} \rightarrow \texttt{Statement}.

Finally, we need functionality for stitching together statements and skipping statements we do not need (for example, we may not want anything inside the body of an ELSE statement). Thus, we have \texttt{sequence} : \texttt{Statement} \rightarrow \texttt{Statement} \rightarrow \texttt{Statement} to create a sequence of statements and \texttt{skip} : \texttt{Statement} to act as a do-nothing statement. Listing 9 demonstrates the \texttt{Statement} data type in its entirety.

Recall from Section 2 that program organization units (POUs) are the building blocks of PLC software. As shown in Listing 10, we define an additional data type, \texttt{POU}, which has members \texttt{program}, \texttt{function}, and \texttt{function block}, each which requires two parameters: a list of variables (as strings) present in the POU and a Statement representing the code body of the POU.
3.2 Formalizing the Semantics of ST in Coq

So far, our model $M$ provides Coq with no understanding of what the syntax we have implemented actually does. For example, $M$ now knows the syntax of a FOR loop but does not know how to actually execute a FOR loop.

To define the semantics of any ST code, $M$ must have a concrete way of representing the environment, or state, of an ST program. It is commonplace to represent the state of a program with all variable, register, and memory values. We are concerned only with the accuracy of the ST program, so we represent our environment state only with the set of all current (variable, value) tuples. Thus, $M$ models the environment state as a list of pairs of strings and values, where the string is the variable name and the value is the variable’s value. In Coq, however, lists can only store data of one type. So far $M$ defines the syntax of both arithmetic and boolean expressions, meaning we need our list to store both int and bool data types. We provide a workaround to this by defining a new data type val, which has two members, int.val and bool.val. Now, we can declare our environment list as Definition Env := list (string * val).

We will next define the semantics of $M$’s arithmetic expressions, ArithExpression. For this, we create a new type, single_step, as shown in Listing 11. The var_step, beginning on Line 2, defines the rules for computing a variable (i.e. reducing a variable to its integer value); it makes use of two helper functions, namely:

1. get, which returns the value $v$ of a given variable $x$ in the environment $env$.

2. set, which sets the variable $x$ to have the value $v$.

The remaining members of single_step define the execution of all allowable arithmetic operations.

We next define the semantics of boolean expressions, as shown in Listing 12. Similar to var_step, var_step_bool tells Coq to simplify a variable holding a boolean value by replacing

---

4 The members beginning with “cong” clarify congruence rules for the operations, e.g. $a + b = b + a$. 

22
**Inferential single_step**: \( (\text{ArithExpression} \times \text{Env}) \rightarrow (\text{ArithExpression} \times \text{Env}) \rightarrow \text{Prop} \)

(* Execution Rule for Variable Assignment *)

| var_step: \( \forall v x \text{ env}, \ \text{get v env} = \text{Some x} \rightarrow \)
| single_step \( (\text{var v, env}) (\text{int x, env}) \)

(* Addition Execution *)

| \( \forall x y \text{ env}, \)
| single_step \( (\text{add (int x) (int y), env}) (\text{int (Z.add x y), env}) \)

(* Congruence Rules for Addition *)

| cong_add_r: \( \forall e1 e2 e2' \text{ env env'}, \)
| single_step \( (e2, env) (e2', env') \rightarrow \)
| single_step \( (\text{add e1 e2, env}) (\text{add e1 e2', env'}) \)

| cong_add_l: \( \forall e2 e1 e1' \text{ env env'}, \)
| single_step \( (e1, env) (e1', env') \rightarrow \)
| single_step \( (\text{add e1 e2, env}) (\text{add e1' e2, env'}) \)

(* Division Execution *)

| \( \forall x y \text{ env}, \)
| \( y \neq 0\%Z \rightarrow \)
| single_step \( (\text{div (int x) (int y), env}) (\text{int (Z.div x y), env}) \)

(* Multiplication Execution *)

| \( \forall x y \text{ env}, \)
| single_step \( (\text{mul (int x) (int y), env}) (\text{int (Z.mul x y), env}) \)

(* Congruence Rules for Multiplication *)

| cong_mul_r: \( \forall e1 e2 e2' \text{ env env'}, \)
| single_step \( (e2, env) (e2', env') \rightarrow \)
| single_step \( (\text{mul e1 e2, env}) (\text{mul e1 e2', env'}) \)

| cong_mul_l: \( \forall e2 e1 e1' \text{ env env'}, \)
| single_step \( (e1, env) (e1', env') \rightarrow \)
| single_step \( (\text{mul e1 e2, env}) (\text{mul e1' e2, env'}) \)

(* Subtraction Execution *)

| \( \forall x y \text{ env}, \)
| single_step \( (\text{sub (int x) (int y), env}) (\text{int (Z.sub x y), env}) \)

---

**Listing 11**: Declaration of type to define the semantics of ST arithmetic expressions in Coq
1 Inductive boolean_single_step : (BooleanExpr * Env) ->
2 (BooleanExpr * Env) -> Prop :=
3 (* Execution Rule for Variable Assignment *)
4 | var_step_bool : forall x v env, get x env = Some v ->
5 boolean_single_step (var_bool x, env)
6 (if in_ v Z then bool_cond false else bool_cond
7 (get_bool_val_bool v), env)
8 (* Execution Rules for Logical Operators *)
9 | compute_not : forall b env, boolean_single_step
10 (not (bool_cond b), env) (bool_cond (negb b), env)
11 | compute_and : forall b1 b2 env, boolean_single_step
12 (and_ (bool_cond b1) b2, env) (if b1 then b2 else bool_cond false, env)
13 | compute_or : forall b1 b2 env, boolean_single_step
14 (or (bool_cond b1) b2, env) (if b1 then bool_cond true else b2, env)
15 (* Execution Rules for Equality Operators *)
16 | compute_equal : forall m n env, boolean_single_step
17 (equal (int m) (int n), env) (bool_cond (Z.eqb m n), env)
18 | compute_lt : forall m n env, boolean_single_step
19 (lt (int m) (int n), env) (bool_cond (Z.ltb m n), env)
20 | compute_lte : forall m n env, boolean_single_step
21 (lte (int m) (int n), env) (bool_cond (Z.leb m n), env)
22 | compute_gt : forall m n env, boolean_single_step
23 (gt (int m) (int n), env) (bool_cond (Z.gtb m n), env)
24 | compute_gte : forall m n env, boolean_single_step
25 (gte (int m) (int n), env) (bool_cond (Z.geb m n), env)
26 (* Congruence Rules *)
27 | cong_le_r : cong_r le int single_step boolean_single_step
28 | cong_le_l : cong_l le single_step boolean_single_step
29 | cong_not : cong_1 not boolean_single_step boolean_single_step
30 | cong_and_ : cong_l and_ boolean_single_step boolean_single_step.

Listing 12: Declaration of type to define the semantics of ST boolean expressions in Coq
the variable with its actual value; we define a function, get_bool_val_bool, to extract the
boolean value of a value \( v \in \text{val} \). The remaining members define the computation of both
logical and relational operators.

Now that \( M \) clearly defines how to evaluate both arithmetic and boolean expres-
sions, we must introduce the semantics of statements to our model. Listing 13 shows the
statement_single_step inductive data type which defines the semantics of each type of
statement.

The next unit up from a statement in ST is a POU. Listing 14 showcases step_POU which
Inductive statement_single_step : (Statement * Env) -> Prop :=
| exec_assign_arith : forall x v v0 env, get x env = Some v0 ->
  statement_single_step (assign_arith x (int v), env) (skip, set x v env)
| cong_assign_arith : forall x,
  cong_1 (assign_arith x) single_step statement_single_step
(* Sequence Execution Rule *)
| exec_sequence : forall s env,
  statement_single_step (sequence skip s,env) (s,env)
(* Sequence Congruence *)
| cong_sequence : cong_1 sequence
statement_single_step statement_single_step
(* IF Execution Rule *)
| exec_if_ : forall b s1 s2 env,
  statement_single_step (if_ (boolean_cond b) s1 s2, env)
  (if b then s1 else s2, env)
(* IF Congruence *)
| cong_if_ : forall b b' env env' s1 s2,
  boolean_single_step (b,env) (b',env') ->
  statement_single_step (if_ b s1 s2,env) (if_ b' s1 s2,env')
(* WHILE Execution Rule *)
| exec_while : forall b s env,
  statement_single_step (while b s,env)
  (if_ b (sequence s (while b s)) skip, env)
(* FOR Execution Rule *)
| exec_for : forall count_var_str j k smnt env,
  statement_single_step (for_ count_var_str j k smnt, env)
  (if_ (le (var count_var_str) (j))
    (sequence (smnt) (sequence (assign_arith (count_var_str)
    (add (var count_var_str) (k)))
    (for_ count_var_str j k smnt)) ) skip, env)
(* REPEAT Execution Rule *)
| exec_repeat : forall s b env, statement_single_step (repeat_ s b, env)
  (if_ (b) skip (sequence s (repeat_ s b)), env)
.

Listing 13: Declaration of type to define the execution of ST statements in Coq
tells Coq how to run a representation of an ST program.

We now have a Model $M$ which defines the syntax and semantics for a significant portion of the ST language. As a result, we are ready to begin using our model to formally verify ST programs.
4 Formally Verifying ST Programs

In this section we provide a number of example ST programs, their representation in $M$, Coq specifications that define their intended behavior, and proofs of these specifications. While many of these example programs are relatively simple and perform tasks that likely would not be performed by a real PLC program (e.g., computing the factorial of a number $n$), the examples serve to show the workflow of using our model to formally verify ST programs. Many of our proofs rely on “running” our programs to completion based on the semantic, or execution, rules defined in Section 3.2. For this, we use several functions, tactics, and types provided in [2].

4.1 Example 1: A Simple Kinetic Energy Program

For our first example, we consider a program designed to calculate the kinetic energy of an object. The kinetic energy (KE) of an object is defined as the energy that object possesses as a result of its motion. KE, as shown in Equation (1), is equal to one half the product of the objects mass, $m$, and squared velocity, $v^{[12]}$.

$$KE = \frac{1}{2}mv^2$$

Writing the ST Program: Listing 15 demonstrates our ST program kinetic that calculates the kinetic energy of an object given its mass and velocity. On line 7, we define the three variables of our program: mass, velocity, and kinetic_energy, all of type INT. On lines 10 and 11, we assign values to mass and velocity, respectively. This program assumes that sensors for reading the mass and velocity of an object have been properly configured and connected to the PLC. The syntax used to read the value of a sensor connected to a PLC depends on the PLC being used, in addition to the environment the PLC program runs in. The syntax used on lines 10 and 11 (e.g., Local:2:I.Data[0].0) assumes a Allen-Bradley Logix 5000™ controller programmed in the Studio 5000 Logix Designer development envi-
Program to calculate the kinetic energy of an object given its mass and velocity

```plaintext
1 /*
2 Program to calculate the kinetic energy of an object given its mass and velocity
3 */
4 PROGRAM kinetic
5 VAR
6     mass, velocity, kinetic_energy : INT;
7 END_VAR
8 // read mass of object using non-retentive assignment operator
9 mass := Local:2:I.Data[0].0;
10 // read velocity of object
11 velocity := Local:2:I.Data[0].1;
12 // calculate kinetic energy
13 kinetic_energy := 1 * mass * velocity * velocity;
14 END_PROGRAM
```

Listing 15: A Structured Text program `kinetic` that calculates the kinetic energy of an object given its mass and velocity

```plaintext
1 Definition kinetic_energy_program (mass velocity : Z) : POU :=
2     program ["mass"; "velocity"; "kinetic_energy"]
3         (sequence (assign_arith "mass" (int mass))
4         (sequence (assign_arith "velocity" (int velocity))
5         (assign_arith "kinetic_energy"
6         (mul (mul (mul (int 1) ("mass") (var "velocity")) ("velocity"))))).
```

Listing 16: Representation of the ST program `st_summation` in our ST model $M$

Translating the ST Program to our Model $M$: The first step in using Coq to prove that this program performs its intended behavior is to translate the program to our model $M$ that we presented in Section 3. Listing 16 shows this translation. On line 2, we use the `program` constructor to create a program with variables “mass”, “velocity”, and “kinetic energy”. On lines 3 and 4, we use the `assign_arith` constructor to assign provided integer values to these variables, respectively. On lines 5 and 6, we use the `mul` operator to calculate the kinetic energy and assign it to the `kinetic_energy` variable.

\footnote{Note that we multiply the product of mass and velocity squared by 1, rather than 0.5 here, because our model does not yet support real numbers}
Writing the Program Specification in Coq: As mentioned in 2.2, at the root of formal methods is creating a model to define the syntax and semantics of a target programming language, in addition to writing the formal specification of that program, which is an unambiguous rule stating how the program should behave. Recall back to Example 2.3, in which we define a proposition stating what a correct version of a factorial function implementation should return. We follow a similar approach for the formal specifications in our work. Listing 17 shows the specification kinetic_spec, which formally defines how our kinetic energy program should behave. The claim of our specification, kinetic_formula_claim, states that for all positive values of mass and velocity, the kinetic energy program will execute each step and, more importantly, finish with a final environment, in which the mass variable is set to mass, the velocity variable set to velocity, and the kinetic_energy variable set to 1 * mass * velocity * velocity.

Proving the Program Adheres to its Specification: The next step in the formal verification process is to prove that the kinetic energy program adheres to its specification. Lemma kinetic_ok in Listing ?? states just this. Specifically, it makes use of the sound definition implemented by the authors of [2]. We will not explain sound in detail, but it
Lemma kinetic_ok : sound step_POU kinetic_spec.

essentially states that the specifications stated in a given formal specification \((\text{sum.spec} \text{ in this case})\) are met following the given execution rules \((\text{step.POU} \text{ in our case})\).

To prove this Lemma, we first apply the proved_sound tactic, which is implemented by the authors of \([2]\). We will not explain in detail the inner workings of this tactic, but essentially, it unrolls our proof into the following state:

\[
\begin{align*}
\forall (x : \text{POU} \times \text{Env}) (P : \text{POU} \times \text{Env} \rightarrow \text{Prop}), \\
\text{kinetic.spec} x P \rightarrow \text{step} \text{step.POU} (\text{trans} \text{step.POU} \text{kinetic.spec}) x P
\end{align*}
\]

We observe that we have a number of variables or hypotheses buried within in terms of our proof. For example, \text{kinetic.spec} has the variables \text{mass} and \text{velocity} that we are interested in and seek to use in our proof. Coq’s \text{destruct} tactic is a powerful tactic for performing case analysis on inductive data types. When used on an entire subgoal, \text{destruct} will destruct any hypotheses in the proof state into its subcomponents. We apply \text{destruct} 1 on the above proof state, yielding:

\[
\begin{align*}
\text{mass}, \text{velocity}: \mathbb{Z} \\
H: 0 < \text{mass} \land 0 < \text{velocity}
\end{align*}
\]

1/1

\[
\begin{align*}
\text{step} \text{step.POU} (\text{trans} \text{step.POU} \text{kinetic.spec}) \\
(\text{program} \left[\text{"mass";} \text{"velocity";} \text{"kinetic_energy"}\right] \\
(\text{sequence} (\text{assign.arith} \text{"mass"} (\text{int} \text{mass}))) \\
(\text{sequence} \\
(\text{assign.arith} \text{"velocity"} (\text{int} \text{velocity}))) \\
(\text{assign.arith} \text{"kinetic_energy"} \\
(\text{mul} \\
(\text{mul} (\text{mul} (\text{int} \text{1}) (\text{var} \text{"mass"}))) \\
(\text{var} \text{"velocity"})))
\end{align*}
\]
As seen above, the hypothesis \texttt{kinetic\_spec}\ x\ P has been deconstructed into its components, \texttt{mass}, \texttt{velocity}, and \texttt{H}. We next apply the tactics \texttt{eapply sstep; [solve[step\_tac]]}. yielding. The \texttt{eapply} tactic in Coq is similar to the \texttt{apply} tactic, except \texttt{eapply} leaves \textit{existential variables} in the proof that must be filled in later; these variables cannot be determined at the current proof state. In our specific case, \texttt{eapply sstep} has applied the \texttt{sstep} constructor\footnote{See \cite{2} for this rule} to our current goal, which has the type \texttt{step\ step\_POU (trans\ step\_POU\ kinetic\_spec)\ x\ P}. This tactic generates a new subgoal for each of the premises in \texttt{step}; the variables of these subgoals are existential and the \texttt{[solve[step\_tac]]} fills them in. \texttt{[solve[step\_tac]]} applies the tactic \texttt{step\_tac} to the proof state repeatedly until no longer possible. \texttt{step\_tac}, another tactic defined in \cite{2}, uses pattern matching to apply our semantic rules to the proof. After \texttt{eapply sstep}, our proof state is as follows:

\begin{verbatim}
  step\_POU
  (program ["mass"; "velocity"; "kinetic\_energy"])
  (sequence (assign\_arith "mass" (int mass))
  (sequence
    (assign\_arith "velocity" (int velocity))
    (assign\_arith "kinetic\_energy"
      (mul
        (mul (mul (int 2) (var "mass"))
        (var "velocity")))

\end{verbatim}
step_tac, therefore, applies the step_POU execution rule to this goal to arrive at the following proof state:

mass, velocity: Z
H: 0 < mass \( \land \) 0 < velocity
1/1
trans step_POU kinetic_spec

(program ["velocity"; "kinetic_energy"]
(sequence (assign_arith "mass" (int mass))
(sequence
(assign_arith "velocity" (int velocity))
(assign_arith "kinetic_energy"
  (mul
   (mul (mul (int 2) (var "mass"))
   (var "velocity"))
   (var "velocity"))),
  (("mass", 0))
  (fun cfg' : POU * Env =>
    cfg' =
    (program [] skip,
      [("kinetic_energy",
        2 * mass * velocity * velocity);
        ("velocity", velocity); ("mass", mass)])

We then apply the tactics run;[reflexivity || assumption ..|]., which brings us to the following proof state:

mass, velocity: Z
H: 0 < mass \( \land \) 0 < velocity
trans step_POU kinetic_spec
  (program [] skip,
   set "kinetic_energy" (1 * mass * velocity * velocity)
   (set "velocity" velocity
    (set "mass" mass
     ["kinetic_energy", 0); ("velocity", 0);
     ("mass", 0)]))
  (fun cfg' : POU * Env =>
   cfg' =
   (program [] skip,
    ["kinetic_energy", 2 * mass * velocity * velocity);
    ("velocity", velocity); ("mass", mass)])

The run tactic, another tactic crafted by the authors of [2], considers all of the semantic rules we define in Section 3.2 and applies the appropriate rules so that the program “executes” entirely. Applying simpl gives:

mass, velocity: Z
H: 0 < mass \ 0 < velocity

33
(fun cfg' : POU * Env =>
cfg' =
(program [] skip,
[("kinetic_energy",
   match mass with
   | 0 => 0
   | Z.pos y' => Z.pos y'~0
   | Z.neg y' => Z.neg y'~0
   end * velocity * velocity); ("velocity", velocity);
("mass", mass)])

We then apply the ddone tactic, which essentially terminates our program. Our proof state:

mass, velocity: Z
H: 0 < mass /\ 0 < velocity
1/1
(program [] skip,
[("kinetic_energy",
   match mass with
   | 0 => 0
   | Z.pos y' => Z.pos y'~0
   | Z.neg y' => Z.neg y'~0
   end * velocity * velocity); ("velocity", velocity);
("mass", mass)]) =
(program [] skip,
[("kinetic_energy",
   match mass with
   | 0 => 0
   | Z.pos y' => Z.pos y'~0
   | Z.neg y' => Z.neg y'~0
   end * velocity * velocity); ("velocity", velocity);
("mass", mass)])
\[ Z.\neg y' \Rightarrow Z.\neg y' \sim 0 \]

end * velocity * velocity); ("velocity", velocity);
("mass", mass)])

Here, we can see that both sides of the equation in our goal are identical. A simple reflexivity tactic completes our proof.

### 4.2 Example 2: A Simple Summation Program

For our second example, we use the same program used in [2]—one that sums the numbers 1 through \( n \). Of course, this article ([2]) implements this program in their toy language IMP; we implement this program in ST.

**Writing the ST Program:** Consider the ST program `st_summation` in Listing 18 that calculates the sum of the numbers 1 through \( n \). On line 6, we define two variables, `sum` and `n`, of type `INT`. On lines 8 and 9, respectively, we initialize `sum` to 0 and `n` to the return value of a function `GetN`, which we assume to be a function that allows for user input for one integer value. On line 10 we begin our WHILE loop which executes until \( n \) equals 0. During each iteration of the loop, `sum` is incremented by `n` (line 11) and `n` is decremented by 1 (line 12).

**Translating the ST Program to our Model \( M \):** Listing 18 presents the representation of `sum_program` in \( M \). We use the Coq keyword `Definition` to define a variable `sum_program` of type `POU` that takes in one argument, `N`, of type `Z`. More specifically, `sum_program` is a `program` (which is a type of `POU`). Recall that we define programs in \( M \) to take in two arguments: a list of strings representing the variables used in the program and a `Statement`. For `sum_program`, as the first argument we pass in `["n"; "sum"]`, as the `st_summation` program takes in these exact two variables. For the program’s statement, we pass in a `sequence`, which is capable of holding multiple statements—a program is a collection of
1 /*
2 Program to add up the numbers 1 through n.
3 */
4 PROGRAM st_summation
5 VAR
6 sum, n : INT;
7 END_VAR
8 sum := 0;
9 n := GetN(); // GetN() is a function to read an n value
10 WHILE NOT n <= 0 DO
11 sum := sum + n; // add n to sum
12 n := n - 1; // decrement n
13 END_WHILE;
14 END_PROGRAM

Listing 18: A Structured Text program st_summation that calculates the sum of the numbers 1 through n

1 Definition sum_program N : POU :=
2 program ["n"; "sum"]
3 (sequence (assign_arith "n" (int N))
4 (sequence (assign_arith "sum" (int 0))
5 (while (not (le (var "n") (int 0)))
6 (sequence (assign_arith "sum" (add (var "sum") (var "n")))
7 (assign_arith "n" (add (var "n") (int (-1)))))
8)).

Listing 19: The representation of our summation program, sum_program in our formally specified model

Statements. On lines 3 and 4 we use the assign_arith syntax to assign values to n and sum. Recall that the assign_arith constructor requires a string and an ArithExpression. As the strings, we pass in the variable names and as the ArithExpressions, we pass in int N and 0 (recall that int x is the ArithExpression for instantiating an integer value in our Model). On line 5, we finally begin our While loop. In our model, while is of type Statement, and its constructor requires a BooleanExpr and a Statement. For the BooleanExpr, we pass in not (le (var "n") (int 0)), which translates to “n is not less than equal or equal to 0”. This means that our While loop will execute until n equals 0. For the Statement, we pass in a sequence on with interior Statements to (1) increment sum by n (line 6) and (2) decrement n by 1 (line 7).
Writing the Program Specification in Coq: Here, we define a proposition (i.e., Prop) called `sum_spec` with the `Inductive` keyword which defines the intended behavior of our summation program.

Listing 20 shows our specification `sum_spec`. Our specification has the following two rules:

1. `sum_claim` states that for all `n` greater than or equal to 0, our program always finishes with all statements fully executed according to their execution rules (i.e., `skip` on line 12) and, very importantly, with an environment where `n` is set to 0 and `sum` to the summation of 1 through `n`, or `((n + 1) * n) / 2`.

2. `sum_loop_claim` specifies the behavior of the While loop more more closely. It considers the program at a state where the code inside of the While loop has already executed 0 or more times. At such a point, the environment `env` is such that `n` will have some
Lemma sum_ok : sound step_POU sum_spec.

Listing 21: Lemma sum_ok stating that our summation program adheres to the formal specification sum_spec.

non-negative value \( x \) and \( \text{sum} \) will have some non-negative value \( s \). The specification states that executing the loop to its completion will finish with the environment \( \text{env} \) having \( n \) set to 0 and \( \text{sum} \) increased from its original value \( s \) to the summation of 1 through the value \( x \).

Proving the Program Adheres to its Specification: In the same manner as Lemma kinetic_ok, Lemma sum_ok in Listing 21 states that the summation program adheres to its formal specification. The proof of this lemma is much longer and more complex than the proof for Lemma kinetic_ok, so we omit it from this work.

4.3 Example 3: A Factorial Program

For our third example, we consider a program for calculating the factorial of a number \( n \). The factorial of \( n \) is defined as the product of \( n \) and all the natural numbers below \( n \). That is \( \text{factorial}(n) = n \times (n - 1) \times (n - 2) \ldots \times 1 \). \( \text{factorial}(n) \) is written as \( n! \).

Writing the ST Program: Listing 22 shows our ST program \text{factorial} that calculates the factorial of a number \( n \). On line 3, we declare variables \( n \) and \( \text{prod} \) of type INT. On line 5, we initialize \( \text{prod} \) to 1. On line 6, we begin our FOR loop; we use an incrementer variable \( \text{count} \) assigned initially to be 2. The FOR loop will execute until \( \text{count} \) surpasses \( n \). On line 7, during each iteration of the loop, we assign \( \text{prod} \) to be the product of its current value and \( \text{count} \), which is incremented during each iteration.

Translating the ST Program to our Model \( M \): In a very similar manner to how we write the summation program using our model, we translate our factorial function using our model rules, as shown in Listing 23.
Listing 22: A Structured Text program `factorial` that calculates the factorial of a number `n`.

```
PROGRAM factorial
VAR
  n, prod : INT;
END_VAR
prod := 1;
FOR count := 2 TO n BY 1 DO
  prod := prod * count;
END_FOR;
END_PROGRAM
```

Listing 23: The representation of our factorial program, `fac_program` in our formally specified model.

```
Definition fac_program n : POU :=
program ["n"; "prod"; "count"]
  (sequence (assign_arith "n" (int n))
   (sequence (assign_arith "prod" (int 1))
    (sequence (assign_arith "count" (int 2))
     (for_ ("count") (var "n") (int 1)
      (assign_arith "prod" (mul (var "prod") (var "count"))))))).
```

Writing the Program Specification in Coq: Listing 24 showcases our formal specification, `factorial_spec`, which unambiguously defines how our factorial program should behave. We construct `factorial_spec` very similarly to the specification for our summation program. It states the following two claims:

1. `factorial_claim`, which says that, given any value `n` greater than or equal to 1, our factorial program will execute all of its steps and finish with an environment in which `n` equals `n`, `count` equals `n + 1`, and `prod` equals the factorial of `n`. Note that to calculate the factorial here, we call the same `factorial` function that we formally verify in Example 2.3. In this manner, we can trust that this factorial implementation returns the correct value.

2. `factorial_loop_claim`, which is similar to `sum_loop_claim` in that it formalizes the behavior for the program beginning at a certain iteration of the for loop and continuing to completion. Specifically, it states that if the program is at a point in execution where
Listing 24: Specification \texttt{factorial\_spec} defining the unambiguous intended behavior of our factorial program

\begin{verbatim}
1 Inductive factorial_spec : Spec (POU * Env) :=
2  | factorial_claim: forall n, 1 <= n ->
3  factorial_spec
4  ( program ["n"; "prod"; "count"]
5   (sequence (assign_arith "n" (int n))
6     (sequence (assign_arith "prod" (int 1))
7     (sequence (assign_arith "count" (int 1))
8     (for_ ("count") (var "n") (int 1)
9      (assign_arith "prod" (mul (var "prod") (var "count"))))))))
10   ), []
11 (fun cfg' => cfg' = (program [] skip,
12   [ ("count", (n + 1)); ("prod", factorial (Z.to_nat n)); ("n", (n))]))
14 | factorial_loop_claim : forall env count n, get "count" env = Some count -> 2 <= count <= n + 1 ->
15 forall p, get "prod" env = Some p ->
16  factorial_spec
17  (program []
18   (for_ ("count") (var "n") (int 1)
19    (assign_arith "prod" (mul (var "prod") (var "count"))))
20    , env)
21 (fun cfg' => fst cfg' = program [] skip /
22  snd cfg' = set "n" n (    
23    (set "prod"
24      (( (factorial (Z.to_nat n)) / (factorial (Z.to_nat p)))
25       (set "count" (n + 1) env)))
27)
\end{verbatim}

\texttt{count} is equal to some \(c\) and \texttt{prod} is equal to some \(p\), then the program will finish with \texttt{count} equal to \(n + 1\) and \texttt{prod} equal to the product of the numbers \(p\) and the product of all the numbers in \(p + 1\) through \(n\).

**Proving the Program Adheres to its Specification:** We define \texttt{Lemma factorial\_ok} which states that our factorial program adheres to its formal specification, \texttt{factorial\_spec}. We do this in a manner identical to \texttt{Lemma kinetic\_ok}. Again, we omit the proof of this Lemma, as it is quite lengthy.
Listing 25: Lemma `factorial_ok` stating that our factorial program adheres to the formal specification `factorial_spec`

5 Conclusion

In this work, we demonstrate the use of formal methods applied to PLC code. Specifically, we use Coq to develop a model of the syntax and semantics of the Structured Text PLC programming language. Using this model, we translate a number of ST programs into their equivalent representation in Coq. We then define a formal specification for each of these programs. Next, we employ the theorem-proving capabilities of Coq in order to prove that the programs adhere to these written specifications. The goal of this work is to further highlight the importance of using formal methods to verify the implementation of PLC programs and present one possible technique for doing so, specifically the use of Coq to prove that ST programs meet a pre-defined specification. The model that we present in this work can be used to translate other ST programs into a Coq environment, and the workflow we demonstrate for forming and proving formal specifications can be applied to verify those programs.

6 Future Work

The formal model $M$ for the ST language that we present in this work is incomplete in that it does not model the entirety of the ST programming language. In the future, we would like to further develop $M$ so that it covers the entirety of the ST language, allowing for the translation of all ST programs. Once we complete $M$, we would like to publish the code online so that the research community can use it. Additionally, we would like to develop a compiler that automatically models ST programs in $M$, as the manual approach we present in this work is imperfect and susceptible to human error. Then, we would like to use the completed model $M$ and this compiler in order to prove properties about more complex ST
programs.

References


