HyPLC: Hybrid Programmable Logic Controller Program Translation for Verification

Luis Garcia
Electrical and Computer Engineering Department
University of California, Los Angeles
Los Angeles, CA, USA
garcialuis@ucla.edu

Stefan Mitsch
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
smitsch@cs.cmu.edu

André Platzer
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA
aplatzer@cs.cmu.edu

ABSTRACT

Programmable Logic Controllers (*PLCs*) provide a prominent choice of implementation platform for safety-critical industrial control systems. Formal verification provides ways of establishing correctness guarantees, which can be quite important for such safety-critical applications. But since PLC code does not include an analytic model of the system plant, their verification is limited to discrete properties. In this paper, we, thus, start the other way around with hybrid programs that include continuous plant models in addition to discrete control algorithms. Correctness properties of hybrid programs can be formally verified in the theorem prover KeYmaera X that implements differential dynamic logic, dL, for hybrid programs. After verifying the hybrid program, we now present an approach for translating hybrid programs into PLC code. The new HyPLC tool implements this translation of discrete control code of verified hybrid program models to PLC controller code and, vice versa, the translation of existing PLC code into the discrete control actions for a hybrid program given an additional input of the continuous dynamics of the system to be verified. This approach allows for the generation of real controller code while preserving, by compilation, the correctness of a valid and verified hybrid program. PLCs are common cyber-physical interfaces for safety-critical industrial control applications, and HyPLC serves as a pragmatic tool for bridging formal verification of complex cyber-physical systems at the algorithmic level of hybrid programs with the execution layer of concrete PLC implementations.

CCS CONCEPTS

• **Computing methodologies** → **Model verification and validation**; • **Computer systems organization** → **Embedded and cyber-physical systems**; • **Software and its engineering** → **Compilers**.

ICCPS '19, April 16–18, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6285-6/19/04…$15.00
https://doi.org/10.1145/3302509.3311036

KEYWORDS

Industrial control, programming languages, formal verification, semantics, compilation

ACM Reference Format:
Luis Garcia, Stefan Mitsch, and André Platzer. 2019. HyPLC: Hybrid Programmable Logic Controller Program Translation for Verification . In *10th ACM/IEEE International Conference on Cyber-Physical Systems (with CPS-IoT Week 2019) (ICCPS '19), April 16–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3302509.3311036

1 INTRODUCTION

There has been an increased emphasis on the verification and validation of software used in embedded systems in the context of *industrial control systems* (ICS). ICS represent a class of cyber-physical systems (CPS) that provide monitoring and networked process control for safety-critical industrial environments, e.g., the electric power grid [1], railway safety [2], nuclear reactors [3], and water treatment plants [4]. A prominent choice of implementation platform for many ICS applications are *programmable logic controllers* (PLCs) that act as interfaces between the *cyber world*–i.e., the monitoring entities and process control–and the *physical world*–i.e., the underlying physical system that the ICS is sensing and actuating. Efforts to verify the correctness of PLC applications focus on the code that is loaded onto these controllers [5–8]. Existing methods are based on model checking of safety properties specified in modal temporal logics, e.g., Linear Temporal Logic (LTL) [9] and Computation Tree Logic (CTL) [10]. However, since PLC code does not include a model of the system plant, such analyses are limited to more superficial, discrete properties of the code instead of analyzing safety properties of the resulting physical behavior.

In this paper, we thus start from hybrid systems models of ICS, in which the discrete computations of controllers run together with the continuous evolution of the underlying physical system. That way, correctness properties that consider both control decisions and physical evolution can be verified in the theorem prover KeYmaera X [11]. The verified hybrid programs can then be compiled to PLC code and executed as controllers. The reverse compilation from PLC code to hybrid programs facilitates verifying existing PLC code with respect to pre-defined models of the continuous plant dynamics.

In this paper, we present HyPLC, a tool that compiles verified hybrid systems models into PLC code and vice versa. Figure 1 depicts a high-level overview of the bidirectional compilation provided by HyPLC. The hybrid models are specified in differential dynamic logic, dL [12–14], which is a dynamic logic for hybrid systems expressed

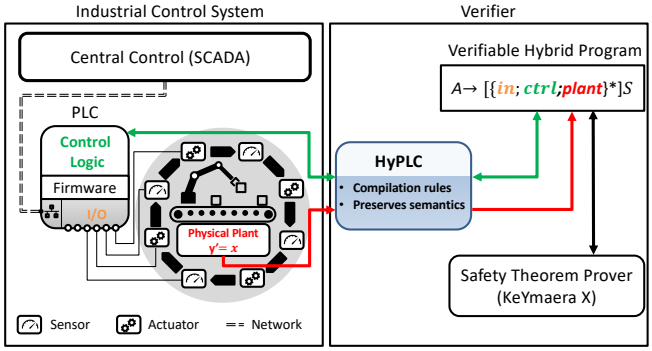

Figure 1: HyPLC provides a bidirectional translation of the discrete control of a verifiable hybrid program expressed in dL and the control logic code that runs on a PLC in the context of a cyber-physical industrial control system

as *hybrid programs*. Compiling hybrid programs to PLC code generates deterministic implementations of the controller abstractions typically found in hybrid programs, which focus on capturing the safety-relevant decisions for verification purposes concisely with nondeterministic modeling concepts. Nondeterminism in hybrid programs can be beneficial for verification since nondeterministic models address a family of (control) programs with a single proof at once, but is detrimental to implementation with Structured Text (ST) programs on PLCs. Therefore, in this paper we focus on hybrid programs in scan cycle form. The compilation adopts the IEC 61131-3 standards for PLCs [15]. Compiling PLC code to dL and hybrid programs, implemented using the ANTLR parser generator [16], provides a means of analyzing PLC code on pre-defined models of continuous evolution with the deductive verification techniques of KeYmaera X. The core contributions of this paper lie in our correctness proofs for the bidirectional compilation, so that both directions of compilation yield a way of obtaining code with safety guarantees (assuming no floating-point arithmetic errors arise). Finally, we evaluated our tool on a water treatment testbed [17] that consists of a distributed network of PLCs.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 introduces compilation rules for terms in both languages and describes how the semantics is preserved. Section 4 and Section 5 describe the compilation of formulas and programs, respectively, and include formal proofs of correctness and preservation of safety across compilation. Section 6 presents our evaluation of HyPLC on a water treatment case study. We discuss the limitations of HyPLC and conclude in Section 7.

2 PRELIMINARIES

This section explains the preliminaries necessary to understand the underlying concepts of HyPLC. We first provide a brief overview of PLCs, including how they are integrated into ICS as well as the associated programming languages and software model as defined by the IEC 61131-3 standard for PLCs [15]. We then discuss previous works in formal verification of PLC programs, followed by an overview of the dynamic logic and hybrid program notation used by HyPLC.

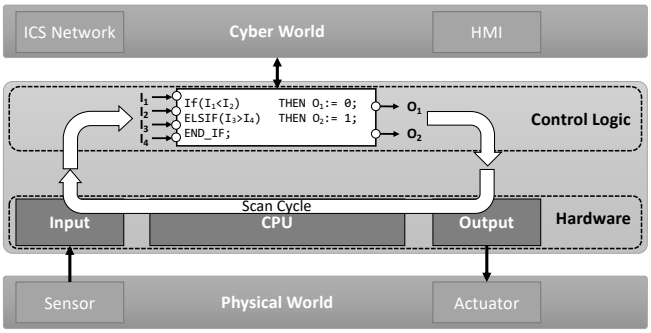

Figure 2: The PLC *scan cycle* in the context of ICS

2.1 Programmable Logic Controllers

Part 3 of the IEC 61131 standards [15] for PLCs specifies both the software architecture as well as the programming languages for the control programs that run on PLCs. We will provide the requisite knowledge for understanding the assumptions made by HyPLC.

PLCs in the context of ICS. Figure 2 shows how PLCs are integrated into ICS as well as a schematic overview of the PLC *scan cycle*. Scan cycles are typical control-loop mechanisms for embedded systems. The PLC "scans" the input values coming from the physical world and processes this system state through the *control logic* of the PLC, which is essentially a reprogrammable digital logic circuit. The outputs of the control logic are then forwarded through the output modules of the PLC to the physical world. HyPLC focuses on hybrid programs of a shape that fits to this scan cycle control principle using time-triggered models.

Programming languages and software execution model. HyPLC focuses on bidirectional compilation of the *Structured Text* (ST) language, which is a textual language similar to Pascal that, for formal verification purposes [18], can be augmented to subsume all other languages[1] defined by the IEC 61131-3 standard. For the software execution model, we refer to the literature [15]. We only consider a single-resource configuration of a PLC that has a single task associated with a particular program that executes for a particular interval, ε. Because, it is a single task configuration, we do not consider priority scheduling.

2.2 PLC Programming Language Verification

Due to their wide use, there have been numerous works regarding the verification of safety properties of PLC programming languages. Rausch *et al.* [19] modeled PLC programs consisting only of Boolean variables, static single assignment of variables, no special functions or function blocks, and no jumps except subroutines without recursion. Such an approach was an initial attempt to provide formal verification of discrete properties of the system, i.e., properties that can be derived and verified purely from the software, ignoring the physical behavior of its plant. Similarly, other approaches have been presented whose safety properties are specified and modeled using linear temporal logic [20, 21] or by representing the system as a finite automaton [22, 23] or real-time automata [24] . The formal

[1] ladder diagrams (LD), function block diagrams (FBD), sequential function charts (SFC), and instruction list (IL)

verification of such systems is limited by state-space exploration techniques, e.g., there will be an uncountable number of states for continuous systems because time is a variable. As such, these techniques will only be able to explore a subset of the states.

Conversely, there have been several works regarding the generation of PLC code based on the formal models of PLC code. PLCSpecif [6] is a framework for generating PLC code based on finite automata representations of the PLC. Although this framework provides a means of generating PLC code based on formally verified models, the formal verification has the aforementioned limitations of providing correctness guarantees for discrete properties of the PLC code that can be verified for a finite time horizon. The approach presented by Sacha [25] has similar limitations since it uses state machines to represent finite-state models of PLC code. Darvas *et al.* also used PLCSpecif for conformance checking of PLC code against temporal properties [26]. Flordal *et al.* automatically generated PLC-code for robotic arms based on generated *zone* models to ensure the arms do not collide with each other as well as to prevent deadlock situations [27]. The approach generates a finite-state model of the robot CPS environment that is then used to generate supervisory code within the PLC that controls its arm. The approach abstracts the PLC's discrete properties and does not incorporate the PLC's timing properties into the physical plant model. Furthermore, this is a domain-specific approach for robot simulation environments and does not provide generalizability nor a means of formal verification of the initially generated finite-state models.

VeriPhy [28] compiles CPS models specified in dL to verified executables that sandbox controllers with safe fallback control and monitor for expected plant behavior. The VeriPhy pipeline combines multiple tools to bridge implementation and arithmetic gaps and provide proofs that safety is preserved when compiling to a controller executable. HyPLC provides bi-directional compilation in the particular context of PLC scan cycles but ignores arithmetic rounding and is not formally verified. Majumdar *et al.* also explored equivalence checking of C code and an associated SIMULINK model [29]. Although such an approach is useful for modelling the behavior of C code in a control system model, additional efforts are needed to interface such a model with verification tools such as KeYmaera X as well as to model the behavior of PLCs.

2.3 Differential Dynamic Logic and Hybrid Programs

HyPLC works on models that have been specified in differential dynamic logic (dL) [12–14], a logic that models hybrid systems and can be formally verified with a sound proof calculus. The formalized models that use dL are referred to as *hybrid programs*. As with ST, we will recall the syntax and semantics of dL and hybrid programs as needed throughout the course of this paper.

The modal operators $[\alpha]$ and $\langle\alpha\rangle$ are used to formally describe the behavioral properties the system has to satisfy. If α denotes a hybrid program, and ϕ and ψ are formulas, then the dL formula

$$\phi \rightarrow [\alpha]\psi$$

means "if ϕ is initially satisfied, then ψ holds true for all the states after executing the hybrid program α". This way, safety properties

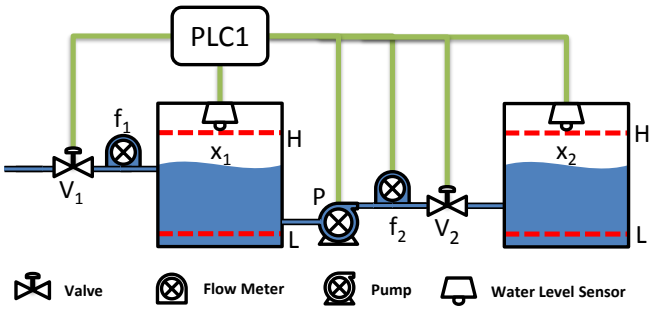

Figure 3: The first process control components for a water treatment testbed [17].

can be encoded for a model α. We use the modeling pattern

$$A \rightarrow [\{\text{ctrl};\text{plant}\}^*]S,$$

where A represents assumptions on the initial state of the system, ctrl describes the discrete control transitions of the system, plant defines the continuous physical behavior of the system, and S is the safety property we want to prove. In this pattern, control and plant are repeated any number of times, as indicated with the nondeterministic repetition operator *.

FV(ϕ) refers to the free variables and BV(ϕ) refers to the bound variables of formula ϕ (accordingly for terms and programs) [13].

2.4 Use Case: Water Treatment Testbed

As a running example, we will use a simple water tank component taken from the first of six control processes of a water treatment testbed [17], depicted in Figure 3. This process is responsible for taking in water from a raw water source and feeding it into a tank. This water will then be pumped out into a second tank to be treated with chemicals. For this first process, the PLC is responsible for controlling the inflow of water for both tanks by opening or closing valves, V_1 and V_2, as well as the outflow of water to the second tank by running the pump, P. The PLC monitors the water level of both water tanks, x_1 and x_2, to ensure that V_1 and V_2, respectively, are closed before each respective tank overflows beyond an upper bound, H. The PLC is additionally responsible for protecting the outflow pump, P, by ensuring that the pump is off if the water level of x_1 is below a lower threshold, L, or if the flow rate of the pump, f_2, is below a certain lower threshold, F_L (not shown in Figure 3). Figure 4 shows a simplified representation of the actual ST code that is loaded onto the PLC for a particular sample rate of ε for all the associated sensors. In this model, the flow rate for the incoming raw water, f_1, is not incorporated into the process control. The real system simply monitors the value of this flow rate without establishing a physical dependency. The upper limits of the water tank level, H_1 and H_2, and the lower limits, L_1 and L_2, represent trigger levels that are below and above, respectively, the actual safety thresholds, H_H and L_L. The trigger values were determined empirically [17]; our proofs will find and verify symbolic characterizations of the trigger values. This model will be used throughout the paper to illustrate how an existing ST program can be systematically compiled to the discrete control of a hybrid program and updated if necessary to ensure safe operation of the ICS.

```
PROGRAM prog0
  /* data declaration */
  VAR_INPUT  x1, x2, f1, f2 : REAL; END_VAR
  VAR_OUTPUT V1, V2, P : BOOL;      END_VAR
  /* structured text program statements */
  IF(x1 >= H1) THEN           V1:=0;     ELSE
  IF(x1 <= L1) THEN           V1:=1;     END_IF;
                                         END_IF;
  IF(x2 <= L2) THEN    P:=1; V2:=1;      END_IF;
  IF(x1 <= LL OR f2 <= FL OR x2 >= H2) THEN
                       P:=0; V2:=0;      END_IF;
END_PROGRAM
/* PLC config and computing task assignment */
CONFIGURATION Config0
  RESOURCE Res0 ON PLC
    TASK Main(INTERVAL:=T#1s, PRIORITY:=0);
      PROGRAM Inst0 WITH Main : prog0;
  END_RESOURCE
END_CONFIGURATION
```

Figure 4: ST program for simplified PLC process control of the system in Figure 3

3 COMPILATION OF TERMS

Compilation approach overview. Compilation between ST and hybrid programs bases on two main ingredients: the syntax of the languages, given in grammars, define their notation; the language semantics give meaning to the syntactic constructs. Compilation translates from one syntax to another, but it must be done in a way that preserves the semantics of the compiled programs.

With compilation rules, we define how to compile a term, formula, or program in the source syntax into a corresponding term, formula, or program of the target syntax. Each rule will compile a certain program operator, and often invoke compilation on the operands. For example, $\mathsf{ST}(\phi \wedge \psi) \triangleright \mathsf{ST}(\phi)$ AND $\mathsf{ST}(\psi)$ compiles conjunction $\wedge$ in hybrid program formulas into conjunction AND in ST of the recursively compiled sub-formulas ϕ and ψ. Here, $\mathsf{ST}(\phi \wedge \psi)$ means that we compile hybrid program formula $\phi \wedge \psi$ into an ST formula; the operator $\triangleright$ describes how the compilation is done.

With proofs of compilation correctness we then show that the compilation rules preserve the semantics *in a way that will allow us to conclude safety of an ST program from a safety proof of a hybrid program.* The proofs exploit the recursive nature of the compilation rules and apply *structural induction* on the program syntax constructs, where we inductively justify each compilation rule from its easier pieces, *assuming absence of arithmetic inaccuracies* and basing on the hypothesis that the easier pieces are correctly built from the base constructs (e.g. complicated terms built from numbers and variables).

For terms and propositional formulas, the compilation rules are straightforward. The main syntactic difference is between nondeterministic choices in hybrid programs and if-then-else constructs in ST. Aligning the semantics in the compilation correctness proofs, however, requires more work: the semantics of ST is given as an operational semantics [18], which describes the effects of taking a step in a program, whereas the semantics of hybrid programs is denotational, which describes the reachability relation of a program.

Term compilation overview. In this section, we will define birectional compilation rules of the arithmetic terms in both hybrid programs and ST for PLCs. The terms of ST are the leaf elements of ST expressions that represent the values stored in the PLC's memory and directly affect the sensing and actuation of the cyber-physical system for a particular context. As such, these values will need to be abstracted to represent the terms of an equivalent hybrid program. We will first discuss syntax of the terms in both languages and then define the semantics-preserving compilation.

Notation. We write $\mathsf{ST}(\theta)$ for the result of compiling a hybrid program term, θ, to an ST term, and we write $\mathsf{HP}(\theta)$ to represent compiling an ST term to a hybrid program term. We write $\mathsf{ST}(\theta) \triangleright s$ when s is the result of compiling θ to an ST term, and $\mathsf{HP}(s) \triangleright \theta$ when θ is the result of compiling s to a dL term. This notation will also be used for the bidirectional compilation of formulas and programs.

3.1 Grammar Definitions

In order to compile terms between both languages while preserving the respective semantics, we first define the grammar for both languages.

Grammar of ST terms. The terms of ST considered in this paper are defined by the grammar:

$$\theta, \eta ::= a \mid x \mid -\theta \mid \theta \sim \eta \text{ where } \sim \in \{+, -, *, /, **\}$$

and where a is a number literal, $x \in V$ is an ST variable, and V is the subset of all ST variables, and both number literals and variables are restricted to `LReal`[2] of the numeric elementary data types defined by the IEC 61131-3 standard.

Grammar of dL terms. The translatable terms of dL and hybrid programs [12, 13] are defined by the grammar:

$$\theta, \eta ::= x \mid n \mid \theta \sim \eta \text{ where } \sim \in \{+, -, \cdot, /, \hat{}\}$$

and where $x \in V$ is a variable and V is the set of all variables. The grammar allows the use of number literals n as functions without arguments that are to be interpreted as the value they represent.

Next, we provide the bidirectional compilation rules of terms and prove term compilation correctness.

3.2 Compilation Rules

We will first define compilation rules for the terminal expressions, referred to as atomic terms, and compose the other expressions following the recursive nature of the grammars.

Atomic terms. Atomic terms in hybrid programs include variables and number literals. For the sake of simplicity, we do not consider functions within hybrid programs as we want to focus on the core elements of discrete control, and *we assume that the data type* `LReal` *of the IEC 61131-3 standard coincides with mathematical reals.* In practice, when a PLC implements `LReal` with floating point numbers, this assumption can be met with an appropriate sound encoding using, for example, interval arithmetic as verified in [28].

HyPLC compiles number literals and variables of hybrid programs, which evaluate to mathematical reals, to numbers and variables of data type `LReal` of the IEC 61131-3 standard as follows: Number literals n and variables x then do not need conversion, so $\mathsf{ST}(n) \triangleright n$ and $\mathsf{HP}(n) \triangleright n$, as well as $\mathsf{ST}(x) \triangleright x$ and $\mathsf{HP(x)} \triangleright x$.

Next, we inductively define the compilation rules for arithmetic operations.

Arithmetic operations. Arithmetic operations are similarly defined in an inductive fashion in similar syntax in both languages,

[2] LReal variables are 64-bit values represented as floating points from the IEC 60559 standard.

which makes translation of terms θ and η straightforward as follows, where $\sim \in \{+, -, /\}$:

$$\mathsf{ST}(-(x)) \triangleright -(\mathsf{ST}(x)) \qquad \mathsf{HP}(-(x)) \triangleright -(\mathsf{HP}(x))$$
$$\mathsf{ST}(\theta \sim \eta) \triangleright \mathsf{ST}(\theta) \sim \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta \sim \eta) \triangleright \mathsf{HP}(\theta) \sim \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta \cdot \eta) \triangleright \mathsf{ST}(\theta) * \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta * \eta) \triangleright \mathsf{HP}(\theta) \cdot \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta \hat{} \eta) \triangleright \mathsf{ST}(\theta) ** \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta ** \eta) \triangleright \mathsf{HP}(\theta) \hat{} \mathsf{HP}(\eta)$$

We now provide the Lemmas for correctness of the translation of terms in both directions. As in [18], we write $(\theta, \nu) \rightarrow_a c$ to express that in ST a term θ evaluates to c in context ν. We write $\nu[[\theta]] = c$ to express that in dL a term θ evaluates to c at state ν [13]. Details on the dL semantics and ST semantics used in the proof can be found in the associated appendices of the full report [30].

Lemma 3.1 (Correctness of term compilation). *Assuming absence of arithmetic inaccuracies in* LReal*: if* $(\theta, \nu) \rightarrow_a c$ *then* $\nu[[\mathit{HP}(\theta)]] = c$*; conversely, if* $\nu[[\theta]] = c$ *then* $(\mathit{ST}(\theta), \nu) \rightarrow_a c$.

Proof. By structural induction on term operators, see the associated appendix in the full report [30]. □

We next define how the compilation of terms is leveraged to compile the *formulas* of both languages in both directions.

4 COMPILATION OF FORMULAS

In this section, we compile modality- and quantifier-free formulas used in tests in hybrid programs and conditional expressions of ST statements. As was done with the terms of each language, we first discuss the syntax of the formulas for both languages.

4.1 Grammar Definitions

Grammar of ST formulas. ST formulas are used in conditional expressions defined by the IEC 61131-3 standard as follows.

$$\phi, \psi ::= \text{TRUE} \mid \text{FALSE} \mid \theta \bowtie_{\mathsf{ST}} \eta \mid \text{NOT}(\phi) \mid \phi \frown_{\mathsf{ST}} \psi$$
$$\text{where } \bowtie_{\mathsf{ST}} \in \{<, >, >=, <=, <>, =\}$$
$$\text{and } \frown_{\mathsf{ST}} \in \{\text{AND}, \text{OR}, \text{XOR}\}$$

The values TRUE and FALSE represent the two Boolean values a conditional expression can take upon evaluation, θ and η are ST terms, operator $\bowtie_{\mathsf{ST}}$ ranges over relational operators used in ST, operator $\frown_{\mathsf{ST}}$ ranges over logical operators between two formulas, and $\text{NOT}(\phi)$ is the logical negation of a formula ϕ.

Grammar of dL formulas. The truncated grammar for modality- and quantifier-free formulas in dL that we consider in this paper is built using propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ [12] as follows:

$$\phi, \psi ::= \mathit{true} \mid \mathit{false} \mid \theta \bowtie_{\mathsf{HP}} \eta \mid \neg\phi \mid \phi \frown_{\mathsf{HP}} \psi$$
$$\text{where } \bowtie_{\mathsf{HP}} \in \{<, >, \geq, \leq, =, \neq\} \text{ and } \frown_{\mathsf{HP}} \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$$

and where θ and η are dL terms. Wwe present the compilation rules and the formula compilation correctness proof for these grammars.

4.2 Compilation Rules

Atomic formulas. Atomic formulas in both languages comprise the literals *true* and *false* and comparisons of terms and are compiled in a straightforward way:

$$\mathsf{ST}(\mathit{true}) \triangleright \text{TRUE} \qquad \mathsf{HP}(\text{TRUE}) \triangleright \mathit{true}$$
$$\mathsf{ST}(\mathit{false}) \triangleright \text{FALSE} \qquad \mathsf{HP}(\text{FALSE}) \triangleright \mathit{false}$$

Comparisons are directly compiled as follows:

$$\mathsf{ST}(\theta = \eta) \triangleright \mathsf{ST}(\theta) = \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta = \eta) \triangleright \mathsf{HP}(\theta) = \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta \neq \eta) \triangleright \mathsf{ST}(\theta) <> \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta <> \eta) \triangleright \mathsf{HP}(\theta) \neq \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta > \eta) \triangleright \mathsf{ST}(\theta) > \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta > \eta) \triangleright \mathsf{HP}(\theta) > \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta \geq \eta) \triangleright \mathsf{ST}(\theta) >= \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta >= \eta) \triangleright \mathsf{HP}(\theta) \geq \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta < \eta) \triangleright \mathsf{ST}(\theta) < \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta < \eta) \triangleright \mathsf{HP}(\theta) < \mathsf{HP}(\eta)$$
$$\mathsf{ST}(\theta \leq \eta) \triangleright \mathsf{ST}(\theta) <= \mathsf{ST}(\eta) \qquad \mathsf{HP}(\theta <= \eta) \triangleright \mathsf{HP}(\theta) \leq \mathsf{HP}(\eta)$$

The compilation rules for the atomic formulas are the basis for compiling compound formulas.

Logical formulas. Logical connectives $\neg, \wedge, \vee$ are straightforward, whereas $\rightarrow, \leftrightarrow$ are rewritten for compilation (similar for XOR):

$$\mathsf{ST}(\neg(\phi)) \triangleright \text{NOT}(\mathsf{ST}(\phi))$$
$$\mathsf{HP}(\text{NOT}(\phi)) \triangleright \neg(\mathsf{HP}(\phi))$$
$$\mathsf{ST}(\phi \wedge \psi) \triangleright \mathsf{ST}(\phi) \text{ AND } \mathsf{ST}(\psi)$$
$$\mathsf{HP}(\phi \text{ AND } \psi) \triangleright \mathsf{HP}(\phi) \wedge \mathsf{HP}(\psi)$$
$$\mathsf{ST}(\phi \vee \psi) \triangleright \mathsf{ST}(\phi) \text{ OR } \mathsf{ST}(\psi)$$
$$\mathsf{HP}(\phi \text{ OR } \psi) \triangleright \mathsf{HP}(\phi) \vee \mathsf{HP}(\psi)$$
$$\mathsf{ST}(\phi \rightarrow \psi) \triangleright \mathsf{ST}(\neg\phi \vee \psi)$$
$$\mathsf{ST}(\phi \leftrightarrow \psi) \triangleright \text{NOT}(\mathsf{ST}(\phi) \text{ XOR } \mathsf{ST}(\psi))$$
$$\mathsf{HP}(\phi \text{ XOR } \psi) \triangleright \neg(\mathsf{HP}(\phi) \leftrightarrow \mathsf{HP}(\psi))$$

We now prove correctness of the compilation of formulas in both directions. In ST, we write $(\phi, \nu) \rightarrow_a \top$ as in [18] and in dL $\nu \models \phi$ to say that formula ϕ is true at state ν as in [13].

Lemma 4.1 (Correctness of formula compilation). *Formulas evaluate equivalently:* $\nu \models \phi$ *iff* $(\mathit{ST}(\phi), \nu) \rightarrow_a \top$ *and, conversely,* $(\phi, \nu) \rightarrow_a \top$ *iff* $\nu \models \mathit{HP}(\phi)$.

Proof. By structural induction on formula operators, see the associated appendix in the full report [30]. □

5 COMPILATION OF PROGRAMS

Now that we know how to correctly compile terms and formulas in both languages, we turn to compiling program constructs. Since these programs, when executed on a PLC, interact with the physical world, our overall goal is to provably establish safety properties of the physical behavior of an ICS. To this end, we again show compilation correctness with respect to the semantics of the languages, which will serve as a stepping stone to describe the program effect in the larger context of the PLC scan cycle.

We first provide an overview of our hybrid system model of a PLC scan cycle, before we introduce the grammars and compilation rules for both languages and prove compilation correctness.

5.1 Scan Cycle Hybrid System Model

We model the PLC scan cycle as a hybrid program of a particular shape—referred to as a hybrid program in *scan cycle normal form*—in order for safety properties verified about a hybrid program to directly transfer to its implementation in ST.

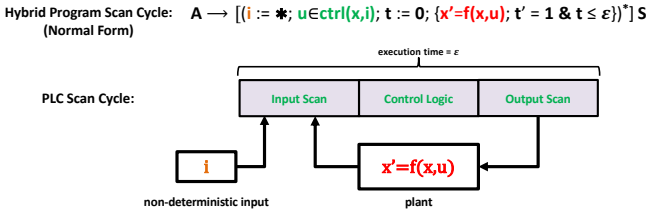

Figure 5: Hybrid system model of a PLC scan cycle

Figure 5 provides an overview of the components of a hybrid program in scan cycle normal form and how they relate to a PLC scan cycle. A PLC scan cycle is a periodic process that, on each iteration, scans the inputs, then executes the control logic to set outputs, and finally forwards outputs to the actuators. The total scan cycle duration in this abstracted model is ε.

Our hybrid program model of such a scan cycle uses nondeterministic assignments $i := *$ to model arbitrary external input to the PLC system, such as sensor values whose state cannot be estimated or user input from a user interface. Based on the current state x and inputs i, the controller $u :\in \text{ctrl}(x, i)$ then chooses control actions u from a set of possible choices. The plant modeled by $t := 0; \{x' = f(x, u), t' = 1 \,\&\, t \leq \varepsilon\}$ continuously evolves the system state x according to the control action u along the differential equations $x' = f(x, u)$ and keeps track of the scan cycle duration bound ε with a clock t to evolve for at most duration ε.

Definition 5.1 (Scan cycle normal form). We call a hybrid program with shape $i := *; u :\in \text{ctrl}(x, i); t := 0; \{x' = f(x, u), t' = 1 \,\&\, t \leq \varepsilon\}$ a program in *scan cycle normal form*. It is safe, if formula $A \longrightarrow [(i := *; u :\in \text{ctrl}(x, i); t := 0; \{x' = f(x, u), t' = 1 \,\&\, t \leq \varepsilon\})^*]S$ is valid.

In the following subsections, we describe how a controller $u :\in \text{ctrl}(x, i)$ is translated into an ST program and its associated configuration. We leave code generation for nondeterministic inputs and physical plant components (e.g., monitors that check model and true system execution for compliance) as future work.

We will use the operational semantics of ST and dynamic semantics of hybrid programs to ensure that the compilation preserves meaning. Additionally, we will use the static semantics of hybrid programs in terms of their bound and free variables to derive configuration information for the PLC code (e.g., distinguish between input and output variables).

5.2 Grammar Definitions

We present the respective grammars for programs in each language.

Grammar of ST programs. ST programs refer to the sequence of statements defined by the IEC 61131-3 standard that form entire ST programs. We consider ST statements s_1 and s_2 as follows:

$$s_1, s_2 ::= x := \theta \mid \text{if } (\phi) \text{ then } s_1 \text{ else } s_2 \mid \text{if } (\phi) \text{ then } s_1 \mid s_1; s_2$$

Where $x := \theta$ is assignment of an ST term θ to variable x, if (ϕ) then s_1 else s_2 is a conditional statement where s_1 is executed if ϕ is true and s_2 is executed otherwise, and $s_1; s_2$ is the sequential composition of ST programs where s_2 executes after s_1 has finished its execution. While Structured Text supports several other control structures such as finitely bounded loops and case-statements, these can be represented as a series of if-then-else statements. The dL grammar is composed in a similar fashion.

Grammar of dL programs. The grammar for PLC-translatable dL hybrid programs is defined as follows.

$$\alpha, \beta ::= x := \theta \mid (?\phi; \alpha) \cup \beta \mid (?\phi; \alpha) \cup (?\neg\phi; \beta) \mid (?\phi; \alpha) \cup ?\neg\phi \mid \alpha; \beta$$

Where $x := \theta$ are assignments of the value of a term θ to the variable x and $(?\phi; \alpha) \cup \beta$ is a guarded execution of α (possible if ϕ is true) and default β (can be executed nondeterministically regardless of ϕ being true or false), $(?\phi; \alpha) \cup (?\neg\phi; \beta)$ is an if-then-else conditional statement, $(?\phi; \alpha) \cup ?\neg\phi$ is an if-then conditional statement without else, and $\alpha; \beta$ is a sequential composition [12, 13]. Given these base grammars for the programs, we now present the compilation rules and the associated correctness proofs that will allow us to conclude safety of ST programs from safety proofs of hybrid programs. Preserving safety will allow us to compile existing ST programs into hybrid programs and analyze their interaction with the physical plant for safety, and conversely compile the controllers of hybrid programs into ST programs for execution on a PLC.

5.3 Compilation Rules

Deterministic assignment. Assignments of terms to variables in hybrid programs represent the core of discrete state transitions in a hybrid system.

The syntax and operational effect of a discrete assignment is the same in both languages, so compilation is straightforward:

$$\text{ST}(x := \theta) \triangleright \text{ST}(x) := \text{ST}(\theta)$$
$$\text{HP}(x := \theta) \triangleright \text{HP}(x) := \text{HP}(\theta)$$

The static semantics of discrete assignments in hybrid programs provides information about input and output variables of the generated ST code: an assignment contributes $\text{BV}(x := \theta) = \{x\}$ to the set of output variables, and $\text{FV}(x := \theta) = \text{FV}(\theta)$ to the set of input variables [13].

Sequential composition programs. The sequential composition of two hybrid programs α and β executes the hybrid program β after α has finished, meaning that β never starts if the program α does not terminate. Sequential composition of ST statements has identical meaning, and so compilation between ST and hybrid programs is straightforward as follows:

$$\text{ST}(\alpha; \beta) \triangleright \text{ST}(\alpha); \text{ST}(\beta) \qquad \text{HP}(\alpha; \beta) \triangleright \text{HP}(\alpha); \text{HP}(\beta)$$

A sequential composition contributes the input and output variables of both its sub-programs: it has output variables $\text{BV}(\alpha; \beta) = \text{BV}(\alpha) \cup \text{BV}(\beta)$ and input variables $\text{FV}(\alpha; \beta) = \text{FV}(\alpha) \cup (\text{FV}(\beta) \setminus \text{MBV}(\alpha))$. Note that the input variables are not simply the union of both sub-programs, since some of the free variables of β might be must-bound, so bound on all paths in α—in $\text{MBV}(\alpha)$—and therefore no longer be free in the sequential composition [13].

Remark 1 (ST Task Execution Timing). *The execution of a series of statements with respect to sequential composition assumes that the statements execute atomically, which is defined in the transition semantics of hybrid programs. We do not model the preemption of*

higher priority tasks as the modeling of the PLC's task scheduling is beyond the scope of this paper and left for future research.

HyPLC assumes that the developer designs a system with multiple tasks such that (1) the execution time of a highest priority task is less than its period and that (2) the total execution of all tasks is less than the period of the lowest priority tasks [31].

Conditional programs. In the translatable fragment of hybrid programs we allow tests to occur only as the first statement of the branches in nondeterministic choices, and we allow only nondeterministic choices that are guarded with tests. A nondeterministic choice between hybrid programs $?\phi;\alpha$ and β executes either hybrid program and is resolved on a PLC by favoring execution of $?\phi;\alpha$ over β in an if-then-else statement. The compilation is defined as follows.

$$\mathsf{ST}((?\phi;\alpha)\cup\beta) \triangleright \text{if } (\mathsf{ST}(\phi)) \text{ then } \mathsf{ST}(\alpha) \text{ else } \mathsf{ST}(\beta)$$
$$\mathsf{ST}((?\phi;\alpha)\cup(?\neg\phi;\beta)) \triangleright \text{if } (\mathsf{ST}(\phi)) \text{ then } \mathsf{ST}(\alpha) \text{ else } \mathsf{ST}(\beta)$$
$$\mathsf{ST}((?\phi;\alpha)\cup?\neg\phi) \triangleright \text{if } (\mathsf{ST}(\phi)) \text{ then } \mathsf{ST}(\alpha)$$

The static semantics combines the input and output variables of both programs: output variables $\mathrm{BV}((?\phi;\alpha)\cup\beta) = \mathrm{BV}((?\phi;\alpha)\cup(?\neg\phi;\beta)) = \mathrm{BV}(\alpha)\cup\mathrm{BV}(\beta)$ and input variables $\mathrm{FV}((?\phi;\alpha)\cup\beta) = \mathrm{FV}((?\phi;\alpha)\cup(?\neg\phi;\beta)) = \mathrm{FV}(\phi)\cup\mathrm{FV}(\alpha)\cup\mathrm{FV}(\beta)$.

Because we only consider loop-free semantics and our fragment only has tests at decision points, we avoid backtracking for tests that would otherwise exist deeper in the programs. Instead, the tests will simply be compiled as nested conditional programs.

ST conditional programs compile to guarded nondeterministic choices in hybrid programs as follows:

$$\mathsf{HP}(\text{if } (\phi) \text{ then } \alpha \text{ else } \beta) \triangleright (?\mathsf{HP}(\phi);\mathsf{HP}(\alpha))\cup(?\neg\mathsf{HP}(\phi);\mathsf{HP}(\beta))$$
$$\mathsf{HP}(\text{if } (\phi) \text{ then } \alpha) \triangleright (?\mathsf{HP}(\phi);\mathsf{HP}(\alpha))\cup?\neg\mathsf{HP}(\phi)$$

Next, we prove compilation correctness that will allow us to transfer safety proofs of hybrid programs to ST programs. We write $(s_1,\nu)\rightarrow(s_2,\omega)$ to say that program s_1 executed in context ν transitions to a new context ω with remaining program s_2 [18]. We write $(\nu,\omega)\in[[\alpha]]$ to say that the final state ω is reachable from the initial state ν by running the hybrid program α [13].

Lemma 5.2 (Correctness of ST to HP compilation). *All states reachable with the ST control program are also reachable by the resulting hybrid program: If $(s_1,\nu)\rightarrow(\mathit{skip},\omega)$ then $(\nu,\omega)\in[[\mathit{HP}(s_1)]]$ for all ν,ω, where skip denotes the end of code for a scan cycle.*

Proof. By structural induction on ST programs from the base case (skip,ν), so $(\nu,\nu)\in[[?\mathit{true}]]$, and induction hypothesis $(s_1,\nu)\rightarrow(\mathsf{skip},\omega)$ then $(\nu,\omega)\in[[\mathsf{HP}(s_1)]]$, see the associated appendix of the full report [30]. □

Lemma 5.3 (Correctness of HP to ST compilation). *All states reachable with the resulting ST control program are also reachable by the source hybrid program: If $(\mathit{ST}(\alpha),\nu)\rightarrow(\mathit{skip},\omega)$ then $(\nu,\omega)\in[[\alpha]]$ for all ν,ω.*

Proof. By structural induction over hybrid programs from the base case (skip,ν) so $(\nu,\nu)\in[[?\mathit{true}]]$, and induction hypothesis $(\mathsf{ST}(\alpha),\nu)\rightarrow(\mathsf{skip},\omega)$ then $(\nu,\omega)\in[[\alpha]]$, see the associated appendix of the full report [30]. □

5.4 Preserving Safety Guarantees Across Compilation

Correct compilation guarantees that safety properties verified for hybrid programs in scan cycle normal form shape are preserved for the runs of translated ST programs. Def. 5.4 expresses how a loop-free ST program is executed repeatedly in the scan cycle of a PLC, connected to inputs, and drives the plant through its results.

Definition 5.4 (Run of ST program). A sequence of states $\sigma_0,\sigma_1,\sigma_2,\ldots,\sigma_n$ is a *run* of ST program s_1 with input (variable vector) i and plant $t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}$ with scan cycle duration ε iff for all $i<n$ the program executes to completion $(s_1,\mu_i)\rightarrow(\mathsf{skip},\nu_i)$ for some program start state μ_i obtained from the previous state σ_i in the run by reading input s.t. $(\sigma_i,\mu_i)\in[[i:=*]]$ and some program result state ν_i driving the plant to the next state σ_{i+1} in a continuous transition of duration at most ε s.t. $(\nu_i,\sigma_{i+1})\in[[t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}]]$.

Def. 5.4 defines how an ST program interacts with the physical world; Def.5.1 says that a hybrid program in scan cycle normal form, $i:=*;u:\in\mathrm{ctrl}(x,i);t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}$, is safe if it reaches only safe states in which S is true when started in states where A is true. We now translate safety to the compiled ST program. Intuitively, a hybrid program is compiled safely to ST when any ST program run that starts in a state matching the assumptions (A) reaches only states where running the plant is safe (S), as expressed in Theorem 5.5.

Theorem 5.5 (Compilation Safety). *If the* dL *formula*

$$A\rightarrow[(i:=*;u:\in\mathit{ctrl}(x,i);t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\})^*]S$$

is valid, and a run $\sigma_0,\sigma_1,\sigma_2,\ldots,\sigma_n$ of $\mathit{ST}(u:\in\mathit{ctrl}(x,i))$ with input i and plant $t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}$ starts with satisfied assumptions $\sigma_0\models A$, then $\sigma_i\models S$ for all i.

Proof. By Lemma 5.3: if $(\mathsf{ST}(u:\in\mathrm{ctrl}(x,i)),\mu_i)\rightarrow(\mathsf{skip},\nu)$ then $(\mu_i,\nu)\in[[u:\in\mathrm{ctrl}(x,i)]]$. Since $\sigma_0,\ldots,\sigma_n$ is a run of $\mathsf{ST}(u:\in\mathrm{ctrl}(x,i))$ in input i and plant $t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}$, we have for all $i<n$ that $(\sigma_i,\mu_i)\in[[i:=*]]$, $(\mu_i,\nu_i)\in[[u:\in\mathrm{ctrl}(x,i)]]$, and

$$(\nu,\sigma_{i+1})\in[[t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}]]\ .$$

Thus, by the semantics of sequential composition [13],

$$(\sigma_i,\sigma_{i+1})\in[[i:=*;u:\in\mathrm{ctrl}(x,i);t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\}]]$$

for all $i<n$. Hence, we conclude $\sigma_i\models S$ for all i by the validity of

$$A\rightarrow[(i:=*;u:\in\mathrm{ctrl}(x,i);t:=0;\{x'=f(x,u),t'=1\ \&\ t\leq\varepsilon\})^*]S$$

□

By Theorem 5.5, an ST program enjoys the safety proof of a hybrid program if our compilation was used in the process (either the hybrid program used in the proof was compiled from the ST program, or the hybrid program was the source for compiling the ST program). Next, we analyze the shape and static semantics of a hybrid program in scan-cycle normal form to extract configuration information.

5.5 Cyclic Control Configuration

ST programs are complemented with a configuration that structures the programs into tasks, assigns priorities and execution intervals to these tasks, and allocates computation resources for the tasks. For a hybrid program in scan-cycle normal form per Def. 5.1

$$(i := *; u :\in \text{ctrl}(x,i); t := 0; \{x' = f(x,u), t' = 1 \,\&\, t \leq \varepsilon\})^*$$

do its shape and static semantics provide essential insight into the required configuration information. The modeling pattern in the scan-cycle normal form is that of a time-triggered repetition, achieved by a clock variable t that is reset to 0 before the continuous dynamics, evolves with constant slope 1, and allows following the continuous dynamics for up to ε time. The combined effect is that the input $i := *$ and control $u :\in \text{ctrl}(x,i)$ are executed at least once every ε time. In the compilation setup, a value for ε must be provided (e.g., with a formula $\varepsilon = n$ as part of the assumptions A in the safety proof) and is taken as the scan cycle configuration of a PLC.

For a single task, we define the compilation of a safety property of a hybrid program in scan-cycle normal form to a task as:

$$\begin{aligned}&\texttt{ST}(A \longrightarrow [(i := *; u :\in \text{ctrl}(x,i); \text{plant})^*]S) \triangleright \\ &\qquad \texttt{Task}(\texttt{ST}(u :\in \text{ctrl}(x,i)), \varepsilon), \\ &\qquad \text{where plant} \equiv t := 0; \{x' = f(x,u), t' = 1 \,\&\, t \leq \varepsilon\}\end{aligned}$$

and $\texttt{Task}(\alpha,\varepsilon)$ is a shorthand defining a task[3] that executes α (here the discrete control $u :\in \text{ctrl}(x,i)$ translated to ST), cyclically with an interval ε. Similarly, we define the converse compilation of a task with an ST program α–whose variables i are of type `VAR_INPUT` from the configuration–and execution time of ε as

$$\begin{aligned}&\texttt{HP}(\texttt{Task}(\alpha,\varepsilon)) \triangleright \\ &\qquad A \longrightarrow [(i := *; \texttt{HP}(\alpha); \text{plant})^*]S \\ &\qquad \textit{given } \text{plant} \equiv t := 0; \{x' = f(x,u), t' = 1 \,\&\, t \leq \varepsilon\}\end{aligned}$$

Since the ST program does not include an analytic plant model, the compiled controller is augmented with the differential equations from a plant given as extra input. The sets of input and output variables determined by analyzing the static semantics of the hybrid program inform the program configuration variable declaration blocks `VAR_INPUT` and `VAR_OUTPUT`, as seen in Figure 4.

Extension to multiple tasks. A future extension to multiple tasks would consider a single configuration of a PLC with a single resource that has a one-to-one mapping of task configurations to ST programs. A designated clock t_n per task keeps track of the associated task's execution interval ε_n. The task execution interval is checked periodically every ε_{sc} times, which represents the scan cycle timing of the PLC. Any task with elapsed clock $t_n \geq \varepsilon_n$ is run (which means that tasks are executed with at most ε_{sc} delay).

6 EVALUATION

Now that we have provided the compilation rules used by HyPLC, we evaluate the tool on a real system. HyPLC was implemented as two module extensions for the KeYmaera X tool: one for each compilation direction[4]. For the compilation of hybrid programs to ST, the compilation rules were implemented on top of the existing KeYmaera X parser written in Scala. Given the abstract syntax tree of a hybrid program, HyPLC generates the associated ST code based on the compilation rules. Any ST code generated by HyPLC was validated using the MATIEC 61131-3 open source compiler [32].

The module for the compilation of an ST program to a hybrid program was implemented as a parser written in Python that was in part generated by the ANTLR v4 parser generator [16].

We next present how HyPLC was evaluated against the water treatment testbed [17].

$$\begin{aligned}&A \longrightarrow [\{\text{in}; \text{ctrl}; t := 0; \{\text{plant} \,\&\, Q\}\}^*]S \\ &A \equiv L_1 \leq x_1 \wedge x_1 \leq H_1 \wedge L_2 \leq x_2 \wedge x_2 \leq H_2 \\ &\quad \wedge V_1 = 0 \wedge V_2 = 0 \wedge P = 0 \\ &\quad \wedge \varepsilon \geq 0 \wedge F_L > 0 \wedge L_L < L_1 \wedge L_L < L_2 \\ &\quad \wedge L_1 < H_1 \wedge L_2 < H_2 \wedge H_1 < H_H \wedge H_2 < H_H \\ &\text{in} \equiv f_1 := *; \; f_2 := * \\ &\text{ctrl} \equiv \{ \;?\; (x_1 \geq H_1); \; V_1 := 0 \\ &\quad \cup ?\neg(x_1 \geq H_1); \; \{ \;?\; (x_1 \leq L_1); V_1 := 1 \\ &\quad\quad \cup ?\neg(x_1 \leq L_1)\} \\ &\quad \}; \\ &\quad \{ \;?\; (x_2 \leq L_2); \; P := 1; \; V_2 := 1 \\ &\quad \cup ?\neg(x_1 \leq L_2)\}; \\ &\quad \{ \;?(x_1 < L_L \vee f_2 \leq F_L \vee x_2 > H_2); \; P := 0; \; V_2 := 0 \\ &\quad \cup ?\neg(x_1 < L_L \vee f_2 \leq F_L \vee x_2 > H_2)\} \\ &\text{plant} \equiv x_1' = V_1 \cdot f_1 - V_2 \cdot P \cdot f_2, \; x_2' = V_2 \cdot P \cdot f_2, t' = 1 \\ &Q \equiv t \leq \varepsilon \wedge x_1 \geq 0 \wedge x_2 \geq 0 \wedge f_1 \geq 0 \wedge f_2 \geq 0 \\ &S \equiv L_L \leq x_1 \wedge x_1 \leq H_H \wedge L_L \leq x_2 \wedge x_2 \leq H_H\end{aligned}$$

Figure 6: Hybrid program generated by HyPLC. This is a compilation of the PLC code from Figure 4

6.1 Use Case: Water Treatment Testbed

In the case study, we first compiled the PLC code from the water treatment testbed shown in Figure 4 into a hybrid program. Formal verification in KeYmaera X showed that this implementation is unsafe. We then updated the generated hybrid program with the necessary assumptions to guarantee the safety of the ICS. Finally, we compiled the fixed hybrid program into PLC code that, by Theorem 5.5, enjoys the safety proof of the hybrid program.

6.1.1 Counterexamples in Existing PLC Code. In order to compile the ST controller into a hybrid program of the water treatment testbed, we provide the continuous plant of the ICS in terms of differential equations, as well as the initial state constraints A. These are combined with the compiled ctrl of the ICS that provides the discrete-state transitions of the system. Finally, we define the safety requirement, S, that ensures that the water tank levels always remain within their upper (H_H) and lower (L_L) thresholds.

Figure 6 shows the full hybrid program generated by HyPLC that incorporates both the compiled ST code as well as the continuous

[3] A task is being used here to abstract the other configuration components of an ST program, i.e., Configurations and Resources. We assume only one configuration and one resource at a time in this paper for a single PLC.

[4] The source code for HyPLC is available at http://HyPLC.keymaeraX.org/

$$
\begin{aligned}
&A \rightarrow [\{\text{in}; \mathbf{ctrl}; t := 0; \{\text{plant } \& \ Q\}\}^*]S \\
&\text{ctrl} \equiv \{ \ ? \ (\boldsymbol{f_1 > (H_H - x_1)/\varepsilon}); \ V_1 := 0 \\
&\qquad \cup ?\neg(\boldsymbol{f_1 > (H_H - x_1)/\varepsilon}); \ \{?(x_1 \leq L_1); V_1 := 1 \cup ?\neg(x_1 \leq L_1)\} \ \} \\
&\qquad \{ \ ?(x_2 \leq L_2); \ P := 1; \ V_2 := 1 \cup ?\neg(x_1 \leq L_2) \ \} \\
&\qquad \{ \ ? \ (\boldsymbol{V_1 \cdot f_1 - V_2 \cdot P \cdot f_2 < (L_L - x_1)/\varepsilon} \vee f_2 \leq FL \vee \boldsymbol{V_2 \cdot P \cdot f_2 > (H_H - x_2)/\varepsilon}); \ P := 0; \ V_2 := 0 \\
&\qquad \cup ?\neg(\boldsymbol{V_1 \cdot f_1 - V_2 \cdot P \cdot f_2 < (L_L - x_1)/\varepsilon} \vee f_2 \leq FL \vee \boldsymbol{V_2 \cdot P \cdot f_2 > (H_H - x_2)/\varepsilon}) \ \}
\end{aligned}
$$

Figure 7: Safe controller with mixed decision conditions for valve and pump actuation based on flow rate and empirical thresholds (replaces the controller of Figure 6, only the control decisions that exposed counterexamples in KeYmaera X are changed; changes are highlighted in boldface)

```
IF (f1 > (HH-x1)/ε) THEN V1:=0;
ELSE
  IF (x1 <= L1) THEN V1:=1; END_IF;
END_IF;

IF (x2 <= L2) THEN P:=1; V2:=1; END_IF;

IF (V1*f1-V2*P*f2 < (LL-x1)/ε OR f2<=FL OR
     V2*P*f2 > (HH-x2)/ε) THEN
  P:=0; V2:=0;
END_IF;
```

Figure 8: ST code fragment compiled from safe ctrl (see Figure 7). The variable ε is a placeholder for the concrete task interval time

dynamics of the water treatment testbed. Intuitively, this model cannot be proven as there are no constraints on the flow rates f_1 and f_2, nor do the guards on actuation enforce such constraints. We use KeYmaera X and the dL proof calculus to find counterexamples for the faulty combinations of operating the valves V_1 and V_2, both for concrete threshold values [33] and the generalized threshold conditions $L_L < L_1 < H_1 < H_H \wedge L_L < L_2 < H_2 < H_H$ of Figure 6. Some representative counterexamples are listed below:

- If $x_1 \geq H_1$ (so $V_1 = 0$) and $x_2 \leq H_2$ (so $V_2 = 1$): without time and flow rate bounds, the pump may drain the first tank when it attempts to protect underflow in the second tank; it may also cause overflow of the second tank.
- If only $V_1 = 1$ is open, the first tank may overflow.
- If both valves are open, either tank may overflow, or the first tank may underflow, depending on the ratio of flow rates.

KeYmaera X finds such counterexamples by unrolling the loop and analyzing paths through the loop body to (i) collect assumptions (e.g., conditions in tests $x_1 \geq H_1$, and effects of assignments $V_1 = 1$ from $V_1 := 1$) and (ii) propagate program effects into proof obligations (e.g., the effect of the flow rate and valves on the water level $x_1' = V_1 \cdot f_1 - V_2 \cdot P \cdot f_2$ is propagated into S). A counterexample consists of sample values for the variables such that the collected assumptions are satisfied but the proof obligations are not. Analyzing these sample values point to potential fixes (e.g., no flow into the first tank $f_1 = 0$ with simultaneous large out flow f_2 indicates that the valve V_2 must be turned off before the first tank drains entirely).

6.1.2 Generating Safe PLC Code. The hybrid program was updated to reflect a safe system that restricts the flow rates by modifying the guard values on the discrete control. Figure 7 shows the updated hybrid program that was proved safe with KeYmaera X. Once verified, HyPLC generates the associated PLC code, listed in Figure 8.

Comparison on real-world data. To illustrate the safety guarantees of our system, we developed a Python script to analyze the sensor and actuation values of 4 days worth of sensor data [33]. We check the values of the sensor data relevant to the process described by our model and instantiate the parameters in the model with the values provided in the dataset. At each time sample, the script checks that the collective system state complies[5] with the expected test-actuation sequences enumerated in our model: the recorded actuator commands for the valves and pump must match the expected command from our model, which is determined by matching the recorded sensor values with the test conditions in the model. For instance, the script records a violation if the condition in `IF (f1 > (HH-x1)/ε) THEN V1:=0;` from Figure 8 is met but the recorded actuation differs from closing the valve. We compared the violations with the original ST program in Figure 4 where, e.g., the corresponding condition reads `IF (x1 >= H1)THEN V1:=0;`. For an illustrative example on a snippet of the real data, please refer to the full report.

Our results revealed that the recorded data did not comply with Figure 8 for 238 instances[6] for the verified code in Figure 8 and 439 instances for the original code in Figure 4 out of 40K possible instances[7]. Note that the verified code allows the system to operate closer to its limits for reasons detailed below, providing a more efficient system operation while enjoying the safety guarantees of the proofs in KeYmaera X.

Upon inspection, most of the violations observed occur during initialization and at the thresholds in oscillating normal system operation [33]. For example, during initialization, the data shows a period where valve V_1 is closed and the tank is drained despite not having reached the lower threshold L_1, see [33, Fig. 4b]. During normal operation, the system slightly overshoots or undershoots the intended limits for discrete switching states, e.g., if the system was supposed to close V_1 when $x_1 \leq L_1$, the system may undershoot L_1. These slight overshoots or undershoots are not allowed in the

[5] The relevant conditions to check and expected control choices can be extracted by proof from a hybrid program using ModelPlex [34].
[6] An instance of a model compliance violation is a range of uninterrupted scan cycles where the recorded data deviates from the expected model.
[7] For 403K samples, the duration of each instance was on average 10 scan cycles.

original ST code, but can be tolerated in the verified model that takes into account flow rates for making decisions.

This study allowed us to not only generate safe PLC code, but to also reveal missing conditions in PLC code that has been evaluated empirically to be safe. We further showed that HyPLC may provide a means of operating a system closer to safety limits while at the same time *provably maintaining crucial safety guarantees.*

7 CONCLUSION AND FUTURE WORK

In this paper, we formalize compilation between safety-critical code utilized in industrial control systems (ICS) and the discrete control of hybrid programs specified in differential dynamic logic (dL). We present HyPLC, a tool for bi-directional compilation of code loaded onto programmable logic controllers (PLCs) to and from hybrid programs specified in dL to provide safety guarantees for hybrid correctness properties of the PLC code in the context of the cyber-physical ICS. We evaluated HyPLC on a real water treatment testbed, demonstrating how HyPLC can be utilized to both verify the safety of existing PLC code as well as generate correct PLC code given a verified hybrid program. Future work will focus on lifting assumptions for PLC arithmetic, support for multiple tasks, as well as support for security analysis. This work serves as a foundation for pragmatic verification of PLC code as well as to understand the safety implications of a particular implementation given complex cyber-physical interdependencies.

ACKNOWLEDGMENTS

This research was sponsored by the U.S. Department of Education under the Graduate Assistance in Areas of National Need Fellowship, the U.S. Department of Energy under the grant number DE-OE0000780, the Air Force Office of Scientific Research (AFOSR) under the grant number FA9550-16-1-0288, as well as the Defense Advanced Research Projects Agency (DARPA) Grant Number FA8750-18-C-0092. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

REFERENCES

[1] M. F. McGranaghan, D. R. Mueller, and M. J. Samotyj, "Voltage sags in industrial systems," *IEEE Transactions on industry applications*, vol. 29, no. 2, pp. 397–403, 1993.

[2] "ABB launches new Pluto programmable logic controller for rail safety applications." [Online]. Available: http://www.abb.com/cawp/seitp202/fa405fb9803dd9eac1258035002f53c0.aspx

[3] B. Kesler, "The vulnerability of nuclear facilities to cyber attack; strategic insights: Spring 2010," *Strategic Insights, Monterey, California. Naval Postgraduate School, Spring 2011*, 2011.

[4] S. Manesis, D. Sapidis, and R. King, "Intelligent control of wastewater treatment plants," *Artificial Intelligence in Engineering*, vol. 12, no. 3, pp. 275–281, 1998.

[5] I. Moon, "Modeling programmable logic controllers for logic verification," *IEEE Control Systems*, vol. 14, no. 2, pp. 53–59, 1994.

[6] D. Darvas, E. Blanco Vinuela, and I. Majzik, "A formal specification method for PLC-based applications," in *15th International Conference on Accelerator and Large Experimental Physics Control Systems*. JACoW, 2015, pp. 907–910.

[7] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*. IEEE, 1999, pp. 106–113.

[8] D. Thapa, S. Dangol, and G.-N. Wang, "Transformation from Petri nets model to programmable logic controller using one-to-one mapping technique," in *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, vol. 2. IEEE, 2005, pp. 228–233.

[9] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing and Verification XV*. Springer, 1995, pp. 3–18.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[11] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer, "KeYmaera X: an axiomatic tactical theorem prover for hybrid systems," in *International Conference on Automated Deduction*. Springer, 2015, pp. 527–538.

[12] A. Platzer, "Differential dynamic logic for hybrid systems." *J. Autom. Reas.*, vol. 41, no. 2, pp. 143–189, 2008.

[13] ——, "A complete uniform substitution calculus for differential dynamic logic," *J. Autom. Reas.*, vol. 59, no. 2, pp. 219–265, 2017.

[14] ——, *Logical Foundations of Cyber-Physical Systems*. Switzerland: Springer, 2018.

[15] K.-H. John and M. Tiegelkamp, *IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids*. Springer Science & Business Media, 2010.

[16] "Antlr." [Online]. Available: https://www.antlr.org/

[17] A. P. Mathur and N. O. Tippenhauer, "SWaT: A water treatment testbed for research and training on ICS security," in *Cyber-physical Systems for Smart Water Networks (CySWater), 2016 International Workshop on*. IEEE, 2016, pp. 31–36.

[18] D. Darvas, I. Majzik, and E. B. Viñuela, "PLC program translation for verification purposes," *Periodica Polytechnica. Electrical Engineering and Computer Science*, vol. 61, no. 2, p. 151, 2017.

[19] M. Rausch and B. H. Krogh, "Formal verification of PLC programs," in *American Control Conference, 1998. Proceedings of the 1998*, vol. 1. IEEE, 1998, pp. 234–238.

[20] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A trusted safety verifier for process controller code." in *NDSS*, vol. 14, 2014.

[21] O. Pavlovic, R. Pinger, and M. Kollmann, "Automated formal verification of PLC programs written in IL," in *Conference on Automated Deduction (CADE)*, 2007, pp. 152–163.

[22] T. Mertke and G. Frey, "Formal verification of PLC programs generated from signal interpreted Petri nets," in *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, vol. 4. IEEE, 2001, pp. 2700–2705.

[23] D. Darvas, E. Blanco Vinuela, and B. Fernández Adiego, "PLCverif: A tool to verify PLC programs based on model checking techniques," in *15th International Conference on Accelerator and Large Experimental Physics Control Systems*. JACoW, 2015, pp. 911–914.

[24] J. Tapken and H. Dierks, "MOBY/PLC–graphical development of PLC-automata," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 1998, pp. 311–314.

[25] K. Sacha, "Automatic code generation for PLC controllers," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2005, pp. 303–316.

[26] D. Darvas, I. Majzik, and E. B. Viñuela, "Conformance checking for programmable logic controller programs and specifications," in *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*. IEEE, 2016, pp. 1–8.

[27] H. Flordal, M. Fabian, K. Åkesson, and D. Spensieri, "Automatic model generation and PLC-code implementation for interlocking policies in industrial robot cells," *Control Engineering Practice*, vol. 15, no. 11, pp. 1416–1426, 2007.

[28] R. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer, "VeriPhy: verified controller executables from verified cyber-physical system models," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 617–630.

[29] R. Majumdar, I. Saha, K. Ueda, and H. Yazarel, "Compositional equivalence checking for models and code of control systems," in *CDC*. IEEE, 12 2013, pp. 1564–1571.

[30] L. Garcia, S. Mitsch, and A. Platzer, "HyPLC: Hybrid programmable logic controller program translation for verification," *CoRR*, vol. abs/1902.05205, 2019.

[31] Rockwell Automation, "Logix5000 controllers, tasks, programs, and routines," 2018. [Online]. Available: https://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm005_-en-p.pdf

[32] M. d. Sousa, "MATIEC-IEC 61131-3 compiler," 2014. [Online]. Available: https://bitbucket.org/mjsousa/matiec

[33] J. Goh, S. Adepu, K. N. Junejo, and A. Mathur, "A Dataset to Support Research in the Design of Secure Water Treatment Systems," in *The 11th International Conference on Critical Information Infrastructures Security (CRITIS)*. New York, USA: Springer, October 2016, pp. 1–13.

[34] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," *Form. Methods Syst. Des.*, vol. 49, no. 1, pp. 33–74, 2016, special issue of selected papers from RV'14.