K-ST: A Formal Executable Semantics of the Structured Text Language for PLCs

Kun Wang, Jingyi Wang, Christopher M. Poskitt, Xiangxiang Chen, Jun Sun, and Peng Cheng

Abstract—Programmable Logic Controllers (PLCs) are responsible for automating process control in many industrial systems (e.g. in manufacturing and public infrastructure), and thus it is critical to ensure that they operate correctly and safely. The majority of PLCs are programmed in languages such as Structured Text (ST). However, a lack of formal semantics makes it difficult to ascertain the correctness of their translators and compilers, which vary from vendor-to-vendor. In this work, we develop K-ST, a formal executable semantics for ST in the $\mathbb{K}$ framework. Defined with respect to the IEC 61131-3 standard and PLC vendor manuals, K-ST is a high-level reference semantics that can be used to evaluate the correctness and consistency of different ST implementations. We validate K-ST by executing 567 ST programs extracted from GitHub and comparing the results against existing commercial compilers (i.e., CODESYS, CX-Programmer, and GX Works2). We then apply K-ST to validate the implementation of the open source OpenPLC platform, comparing the executions of several test programs to uncover five bugs and nine functional defects in the compiler.

Index Terms—Formal executable semantics, PLC programming, Structured text, $\mathbb{K}$ framework, OpenPLC.

1 Introduction

Programmable Logic Controllers (PLCs) are responsible for automating process control in several modern industrial systems, e.g. in manufacturing and public infrastructure. It is critical to ensure that PLCs are operating correctly, as any functional or security-related defects may lead to serious incidents in the system. This has most famously been demonstrated by the Stuxnet worm [1], while many other less-known safety and security incidents [2], [3], [4] and potential hazards [5], [6] related to PLCs have resulted in significant consequences, with an estimated $350,000 in damage on average [7].

The majority of PLCs are programmed using languages defined in the IEC 61131-3 open international standard [8]. Programs can be written in graphical languages such as Function Block Diagrams (FBD), but the standard also defines Structured Text (ST), a fully textual language based on the idea of organizing code into 'function blocks' and designed with a syntax similar to Pascal. ST is a particularly important IEC 61131-3 language given its utility for data processing [9], and the fact that snippets of ST are actually required in FBD and other graphical languages. It is therefore important that translators and compilers for ST are correctly implemented and exhibit only expected behaviors when the code is being run on a PLC.

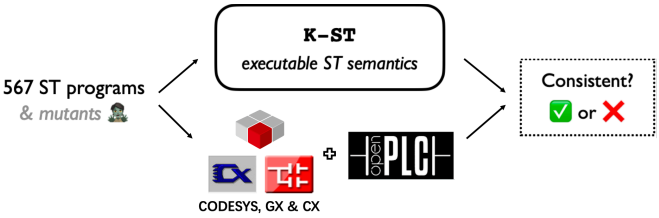

Fig. 1: High-level workflow of our approach

This has motivated a surge of research on analyzing and verifying PLC programs [7], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], although few works focus on ST implementations/compilers. Zhang et al. [24] propose VetPLC, a temporal context-aware, program-based approach to produce timed event sequences that can be used for automatic safety vetting. McLaughlin et al. [21] propose TSV which translates assembly-level code into an intermediate language (ILIL) to verify safety-critical code executed on PLCs. Mader and Wupper [26] translate Instruction List (IL) code into timed automata [31]. Bauer et al. [25] similarly use timed automata as the formalism for Sequential Function Chart (SFC). In [27], the proposed method transforms IL to Petri-nets [32], and manually builds two additional Petri-nets for modeling the PLC and its environment. Xiong et al. [23] propose an algorithm based on variable state analysis for automatically extracting a Behavior Model (BM) from an ST program. These works attempt to transform PLC programs into an intermediate language or another programming language (i.e., C) which is suitable for verifying or detecting potential issues using existing associated verifiers or checkers. The issue of these approaches is that they *lack analysis and proof of equivalence in the conversion process.* In

- *K. Wang, and J. Wang are with the College of Control Science and Engineering, Zhejiang University, Zhejiang 310027, China. E-mail: {kunwang_yml, wangjyee}@zju.edu.cn.*
- *CM. Poskitt is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: cposkitt@smu.edu.sg.*
- *X. Chen is with the College of Control Science and Engineering, Zhejiang University, Zhejiang 310027, China. E-mail: chenxiangx@zju.edu.cn.*
- *J. Sun is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: junsun@smu.edu.sg.*
- *P. Cheng is with the College of Control Science and Engineering, Zhejiang University, Zhejiang 310027, China. E-mail: lunarheart@zju.edu.cn.*

(Corresponding authors: Jingyi Wang and Peng Cheng)

addition, the analyses they perform are often limited (since the existing tools are not designed for PLCs) and do not offer the feedback to the level of source code. Canet et al. [22] propose formal semantics for a significant fragment of the IL language, and a direct coding of this semantics into a model checking tool. Huuck [28] develops a formal operational semantics and abstract semantics for IL, which allows approximating program simulation for a set of inputs in one simulation run. Blech et al. [10], [11], [30] attempted to define the formal semantics of the IL and SFC languages in Coq and NuSMV and, based on that, verify the safety properties in the code. However, IL is a low level assembly-like language that has been deprecated from the IEC61131-3 standard. Furthermore, these studies mainly concentrate on analyzing the functional aspects of the programs and may overlook potential vulnerabilities and security risks introduced during the compilation process.

While extensive research has been conducted on testing more 'traditional' compilers (e.g. vulnerability detection for GCC and Clang [33], [34], [35]), compilers for PLC languages such as ST have received much less attention. The challenges associated with testing the implementation of a compiler arise from the inherent difficulties of ensuring its correctness. One particular challenge stems from the absence of a precise specification of the expected behavior of a compiler. For most popular programming languages, there exist multiple purportedly equivalent implementations of compilers. Compiler testing can take advantage of this by utilizing these implementations as oracles for conducting differential testing [36]. However, in the case of the domain-specific ST language, there is no specific implementation standard, and different vendors often develop their own compilers based on their specific requirements. Another challenge is the semantic complexity of the input and output languages that compilers handle. The fact that different vendors develop their own implementations further exacerbates this issue. Compiler testing methods based on formal semantics [37] have shown advantages in addressing these challenges. With a formal semantics of the ST language, the expected behavior of ST compilers can be precisely and unambiguously defined, which can greatly aid in testing and verifying their correctness.

To the best of our knowledge, a practical and complete semantics for the ST language does not exist, which makes it difficult to ascertain the correctness of ST translators and compilers (e.g. by comparing executions). There are a number of reasons why such a reference semantics is yet to emerge. First, there is insufficient documentation defining or describing the complete features of the ST language [9]. For instance, the official documentation introduces language features by only a few examples, based on which it is difficult for readers to fully understand the behavior of the language. Second, the ST compilers provided by different vendors (e.g. Allen-Bradley, Siemens) can implement the language differently, and their closed source solutions make it difficult to fully assess how they behave systematically (other than through manual observation). For example, CODESYS, CX-Programmer, and GX Works2 all produce negative numbers in the results of negative modulo operations, even though this behavior is undefined according to the official documentation. Furthermore, GX Works2 supports only 10 basic data types, whereas CODESYS supports 17 types. Thus, a formal semantics needs to be 'concrete' enough to be useful, but 'high-level' enough to be general/extendable to the different nuances of vendors' compilers. A preliminary attempt at defining a high-level semantics for ST was made by Huang et al. [38]. However, it falls short of a full reference semantics as it misses several important features of the language, e.g. certain data types, and key sentences.

In this work, we develop K-ST, a formal executable reference semantics for ST in the $\mathbb{K}$ framework [39]. Our high-level semantics is both executable and machine readable, and can be used by the $\mathbb{K}$ framework to generate interpreters, compilers, state-space explorers, model checkers, and deductive program verifiers. Our principal goals for the design of K-ST are as follows:

1) **Validated reference semantics.** K-ST is designed to cover all the main features of ST, and is validated against hundreds of different real-world ST programs extracted from GitHub.
2) **General and extendable.** The semantics is high-level (rather than tied to a particular compiler), with the goal of supporting different ST implementations as well as extensions for vendor-specific functions.
3) **Analyses of ST compilers.** Most importantly, K-ST can be used to check the correctness and consistency of different ST implementations, and thus ensure that a compiler is not introducing an unintended behavior or compile-time threat [40], [41] into a critical industrial system.

Given the absence of complete feature descriptions for the ST language in official documentation, we not only refer to the definitions and code samples in the official documents, but also extensively consult the guidance manuals provided by multiple vendors to better define the semantics of the ST language. For example, there is no specific documentation on how integer overflow is handled in the official documents. Through investigating multiple instruction manuals, we found that existing ST compilers generally use truncation to handle integer overflow without any warning. In defining the semantics, we find that the rewriting rule of the $\mathbb{K}$ framework provides a good mechanism for capturing the unique features of ST. For example, we can rewrite `REPEAT` to `WHILE` to achieve the execution effect of `REPEAT`.

We validate K-ST by extracting 567 real-world ST code samples from GitHub and comparing their executions in our semantics against their executions resulting from various commercial compilers (i.e., CODESYS, CX-Programmer, and GX Works2). We find that K-ST is sufficiently complete to support 509 of these programs (consisting of 26,137 lines of code) and executes those programs correctly (i.e., producing the same outputs as the corresponding existing compiler), with the remaining programs only unsupported due to the use of certain vendor-specific or hardware-related functions that we did not yet formalize. Furthermore, to evaluate the utility of K-ST for testing ST compilers, we compared the executions of the 567 programs (and several mutants) under K-ST and OpenPLC [42], a popular open source PLC program compiler. Through this semantics-based testing, we are able to uncover five bugs and nine functional defects in

the OpenPLC compiler, all of them are previously unknown. Fig. 1 summarises the high-level workflow of this process.

In summary, we make three main contributions.

- We propose an executable formal reference semantics for ST;
- We collect a set of 567 complete ST program samples, and validate the correctness of our executable semantics by running those programs in the semantics and via existing compilers (CODESYS, CX-Programmer, and GX Works2), comparing the results.
- We test OpenPLC, an open source PLC program compiler, using our proposed semantics, and find five bugs and nine functional defects.

The remaining part of this paper is organized as follows. Section 2 introduces the background of ST and the $\mathbb{K}$ framework. The proposed executable operational semantics of ST formalized in $\mathbb{K}$ is introduced in Section 3. Section 4 shows some practical applications of our formal semantics. The evaluation results of the proposed semantics are introduced in Section 5. Section 6 summarises some related work, and Section 7 concludes this work.

2 Background

In this section, we briefly introduce the background of the Structured Text (ST) language and the $\mathbb{K}$ framework.

2.1 Structured Text

The Programmable Logic Controller, invented in 1969 by Dick Morley, is specially designed for applications in industrial environments, e.g. assembly lines, robotic devices, or public infrastructure. These kinds of applications all require high reliability and ease of programming.

Early PLCs were represented as a series of logic expressions in some kind of Boolean format. With the development of programming terminals and the complexity of existing control procedures, Ladder Diagrams (LD) were developed to program PLCs. As of 1993, the IEC 61131-3 standard developed by the International Electrotechnical Commission (IEC) defined five programming languages, including two textual programming languages—ST and IL—as well as three graphical languages—LD, FBD, and SFC. A simple example in Fig. 2 [43] shows a ST code example which can be used for linear scaling of an analog sensor signal.

ST is a high-level PLC programming language which is similar to Pascal [44] (widely used from 1980 to 2000), C/C++ and Java. While it contains common constructs from modern programming languages such as `FUNCTION`, `IF/ELSIF/ELSE` and `CASE` branches, `WHILE` and `FOR` loops, it has its own characteristics, such as the lack of recursion, capitalized keywords, `REPEAT` statement, and `FUNCTION_BLOCK` structure. For instance, `FUNCTION_BLOCK` as an important part of ST, and has its own state. Its main purpose is to modularize and structure a straightforwardly defined portion of the program. It is similar to the class-object manifestation in object-oriented programming. Function blocks exist in two forms: as a type or as an instance, but only the instance can be called. For each function block, the local variables retain their values between each 'call'. TABLE 1 shows the common elements of ST.

```
1  FUNCTION_BLOCK Scale
2      VAR_INPUT
3          ValueIn : REAL;
4          ScaleInMin : REAL;
5          ScaleInMMax : REAL;
6          ScaleOutMin : REAL;
7          ScaleOutMMax : REAL;
8      END_VAR
9      VAR
10         a : REAL;
11         b : REAL;
12         Error : BOOL := FALSE;
13     END_VAR
14     IF ScaleOutMin >= ScaleOutMax THEN
15         Error := TRUE; END_IF;
16     IF ScaleInMin >= ScaleInMax THEN
17         Error := TRUE; END_IF;
18     IF ValueIn < ScaleInMin OR ValueIn > ScaleInMax THEN
19         Error := TRUE; END_IF;
20     IF Error = FALSE THEN
21         a := (ScaleOutMax - ScaleOutMin) / (ScaleInMax - ScaleInMin);
22         b := ScaleOutMax - (a * ScaleInMax);
23         Scale := a * ValueIn + b;
24     ELSE
25         Scale := 0;
26     END_IF;
27 END_FUNCTION_BLOCK
```

Fig. 2: An ST programming example

ST, as the only textual programming language supported by the new IEC standard, has a number of advantages compared to other PLC languages. First, being textual, ST programs can be copied relatively easily. Second, compared with the other four languages, it is more convenient for mathematical calculations, formulas and algorithms, and for managing large amounts of data [9]. Third, compared with 20 years ago, PLC solutions are more in demand today and ST can better adapt to this change. Finally, LD, SFC and FBD also require parts of the program to be written in ST anyway [45], [46].

Unfortunately, the absence of documents defining or describing the complete features of the ST language and the differing customizations of vendors can lead to inconsistent implementations of ST. In addition, understanding the semantics of the ST language, and ensuring that it is formally defined is difficult for end users accustomed to graphical programming. A formal executable semantics of ST not only provides a standard, but also helps PLC engineers verify the completeness and correctness of these implementations.

2.2 The $\mathbb{K}$ Framework

$\mathbb{K}$ is a formal logic framework based on rewriting logic [47]. It was developed with the overarching goal of pursuing the ideal language framework, where all programming languages have formal semantic definitions and all language tools are automatically derived in a correct-by-construction manner at no additional cost. The $\mathbb{K}$ backends, such as the

TABLE 1: Common elements of the ST language

Type	Element	Type	Element	Type	Element
Program Organization Unit	FUNCTION_BLOCK	Built-in Data Type	INT	Built-in Data Type	ARRAY
	FUNCTION		DINT		...
	PROGRAM		SINT	Declaration Type	VAR_GLOBAL
Main Statement	IF		LINT		VAR
	CASE		UINT		VAR_INPUT
	WHILE		UDINT		VAR_OUTPUT
	FOR		USINT		VAR_IN_OUT
	REPEAT		ULINT		VAR_EXTERNAL
	EXIT		REAL		VAR_TEMP
	RETURN		LREAL		AT
	...		BOOL		RETAIN
User Data Type	ENUM		STRING		PERSISTENT
	STRUCT		WSTRING		CONSTANT
Built-in Data Type	TIME		TIME_OF_DAY		...
	DATE		DATE_AND_TIME	...	

Isabelle theory generator, the model checker, and the deductive verifier, can be utilized to prove properties based on the semantics and generated verification tools [48]. Several executable semantics in $\mathbb{K}$ have been developed for mainstream programming languages, including C [49], Java [50], JavaScript [51], Rust [52], Solidity [53], and IMP [54].

A language semantics definition in $\mathbb{K}$ consists of three parts: the language syntax, the configuration, and a set of semantics constructed based on the syntax and the configuration. Given the semantics definition for a programming language and some source programs, $\mathbb{K}$ executes these programs like a translator. For illustration, in the following we take a strict subset of the ST language, i.e., ST_{demo} shown in Fig. 2 as an example to illustrate how to define language semantics in $\mathbb{K}$.

Configuration. The whole configuration cell T of ST_{demo} contains two cells, namely k and $state$. The cell k is used to store the source program $\$PGM$ for execution, and the cell $state$ is used to record the mapping from a variable identifier to its value. The configuration simulates the memory status and environmental changes during runs of the program.

$$\langle \langle \$PGM : Pgm \rangle_k \ \langle .Map \rangle_{state} \rangle_T$$

With the configuration defined, we present the syntax of ST_{demo} in Fig. 3, which includes some numerical operations, logic operations and commonly used statements. Based on the configuration and the syntax of ST_{demo}, we introduce some basic rules in the semantics. The role of the semantics is to tell $\mathbb{K}$ how to execute the source code, where $\mathbb{K}$ executes the code and updates the configuration sentence-by-sentence after parsing the source program.

Here, we show the semantics of $Allocate$, $Lookup$ and $Assignment$ in Fig. 4 as they are the most commonly used constructs in programming languages. TABLE 2 describes some common semantic notations. Take $Allocate$ as an example: when $\mathbb{K}$ runs to lines 9–13 in Fig. 2, the content in the k cell is $\langle$`VAR a : REAL;` VBs `END_VAR` $\cdots\rangle_k$,

```
1  syntax FunctionBlock ::= "FUNCTION_BLOCK" FBBody "END_FUNCTION_BLOCK"
2  syntax FBBody ::= VariableDeclarations Statements
3  syntax VariableType ::= "REAL" | "BOOL"
4  syntax VariableDeclaration ::= "VAR_INPUT" VarBodys "END_VAR"
5                               | "VAR" VarBodys "END_VAR"
6  syntax VarBody ::= Id ":" VariableType ";"
7                               | Id ":" VariableType ":=" Statement ";"
8  syntax Statement ::= Float | Bool
9                               | Statement "*" Statement ";"
10                              | Statement "/" Statement ";"
11                              > Statement "+" Statement ";"
12                              | Statement "-" Statement ";"
13                              > Statement ">=" Statement ";"
14                              | Statement "<" Statement ";"
15                              | Statement ">" Statement ";"
16                              | Statement "=" Statement ";"
17                              > "IF" Statement "THEN" Statements "END_IF;"
18                              | "IF" Statement "THEN" Statements "ELSE" Statements "END_IF;"
19                              > Statement ":=" Statement ";"
20 syntax VariableDeclarations ::= List {VariableDeclaration, ""}
21 syntax Statements ::= List {Statement, ""}
22 syntax VarBodys ::= List {VarBody, ""}
```

Fig. 3: The syntax of ST_{demo}

where VBs stands for `b : REAL; Error : BOOL := FALSE;`. Then, $\mathbb{K}$ will rewrite $\langle$`VAR a : REAL;` VBs `END_VAR` $\cdots\rangle_k$ to $\langle$`VAR` VBs `END_VAR` $\cdots\rangle_k$, which means that `a : REAL;` has been executed according to **rule Variable_Allocate**. Meanwhile, it adds the mapping between the variable name and the corresponding value $(a \mapsto 0.0)$ in the current $state$ cell Rho. In addition, "$requires\ notBool\ (X\ in\ keys\,(Rho))$" guarantees that the variable will not be re-declared. Similarly, variables b and $Error$ will be allocated separately. After that, the content in the k cell is $\langle$`VAR` $.VarBodys$ `END_VAR` $\cdots\rangle_k$, where $.VarBodys$ represents an empty variable declaration list, that is, no additional variable needs to be allocated. The **rule Variable_Finish_Allocate** will be called to convert

rule Variable_Allocate

$$\left\langle \frac{\text{VAR X : REAL; } VBs : VarBodys \text{ END\_VAR}}{\text{VAR } VBs \text{ END\_VAR}} \ \cdots \right\rangle_k$$
$$\left\langle \frac{Rho : Map}{Rho\ (X \longmapsto 0.0)} \right\rangle_{state}$$
$$requires\ notBool\ \ (X\ in\ keys\ (Rho))$$

rule Variable_Finish_Allocate

$$\left\langle \frac{\text{VAR } .VarBodys \text{ END\_VAR}}{.} \ \cdots \right\rangle_k$$

rule Variable_Lookup

$$\left\langle \frac{\text{X} : Id}{I} \ \cdots \right\rangle_k \langle \cdots\ X \longmapsto I\ \cdots \rangle_{state}$$

rule Variable_Assignment

$$\left\langle \frac{\text{X := I : } Float;}{.} \ \cdots \right\rangle_k \left\langle \cdots\ \frac{X \longmapsto \_}{X \longmapsto I} \ \cdots \right\rangle_{state}$$

Fig. 4: The partial semantics of ST_{demo}

TABLE 2: Summary of semantic notations

Notation	Description
rule	The beginning of a semantic rule.
$a \Rightarrow b$	The symbol $\Rightarrow$ means "rewritten by", thus $a \Rightarrow b$ denotes that a can be replaced by b.
$a\ requires\ b$	Execute a when b is true.
$\langle \frac{a}{b} \rangle_k$	$\langle\rangle_k$ stands for the k cell in a configuration.
	Similar to $a \Rightarrow b$, $\frac{a}{b}$ means a will be rewritten by b. However, it can only be used inside $\langle\rangle$.
$\langle \cdots\ a\ \cdots \rangle$	$\cdots$ represents the content in the a context.
.	. stands for empty.
_	Any value.
$a\ :\ b$	The type of variable a is b.
$a \mapsto b$, $a \leftarrow b$	Mapping from a to b.
$a\ \curvearrowright\ b$	The execution of a, followed by execution of b.

"VAR $.VarBodys$ END_VAR" in k to ".", which means that there is no more code to execute in the VAR block and $\mathbb{K}$ will continue to execute the subsequent code.

3 Formal Semantics of Structured Text in The $\mathbb{K}$ framework

In this section, we introduce K-ST, the executable operational semantics of ST formalized in $\mathbb{K}$. Note that in practice the PLC programming environment is provided by specific PLC manufacturers including CODESYS and Siemens's TIA portal (TIA, Structured Control Language (SCL)). As a consequence, the implementations of different manufacturers can vary and may also include their own unique functions or structures.

Our approach is therefore to focus on the common features, allowing other unique functions of the environment to be implemented by extending the operational semantics. Specifically, the syntax of ST is constructed based on the official IEC 61131-3 standard [46]. The configuration is specifically designed for ST. Based on the syntax and the configuration, we then formalize the semantic rules for the language features with rewriting logic. Next, we present each component of the semantic one by one.

3.1 The Syntax of ST

TABLE 3 presents the syntax of ST defined in K-ST, which covers most of the core syntax. We remark that TABLE 3 only contains the main part of K-ST while omitting others, e.g., some built-in functions (`LEN`, `DELETE` and so on) for space reasons. The syntax is specified by a grammar in a dialect of Extended Backus-Naur Form (EBNF) [55], where * means zero or more repetitions. In ST, the top-level grammatical structures include user-defined types (`TYPE` statements) and three Program Organization Units (POUs): `FUNCTION`, `FUNCTION_BLOCK` and `PROGRAM`. Other syntactical elements are derived within these top-level grammatical structures.

3.2 The Configuration of ST

The execution of an ST program needs to update the following kinds of state: data segment, code segment and stack. Among them, the data segment is used to store global variables, the code segment is used to store program execution code, and the stack is used to store local variables of the program. Note that runtime environment switching caused by function calls is also achieved by the operation of stack. The overall runtime configuration of ST in $\mathbb{K}$ is presented in Fig. 5. We highlight our careful design choices as follows.

Overview. There are 11 main cells in the configuration T, i.e., k, $control$, $allenv$, $genv$, $gvenv$, $store$, $type$, $constant$ $input$, $output$ and $nextLoc$. The value of each cell is initialized according to its specified type. For instance, for cells with a mapping relationship, their values are initialized to Map type, and for cells that store a collection, they are initialized to $List$ type. A '.' followed by any type means an empty set of this type. For instance, $.Map$ in the cell $genv$ represents that $genv$ is initialized with an empty map.

Enumeration type. By default, when an enumeration type is defined in ST, PLC compilers will automatically associate a number (indexed from 0 and incremented by 1 each time) to each variable in the enumeration. For repeated declarations, we use $count$ cell to record the value of the current enumeration.

Global variables. There are two types of global variables. First, the POUs and customized types that users define. These variables can be accessed anywhere in the program. We store these variables in the $genv$ cell as the basis for program operation. Second, the variables defined in `VAR_GLOBAL`. These variables cannot be directly accessed in the program unless they are declared with `VAR_EXTERNAL`. We store these variables in the $gvenv$ cell and provide them on demand.

Program execution. The source code parsed by the syntax $SourceUnit$, called $\$PGM$, is stored in the cell k for execution. Then the $\$PGM$ will be executed unit by unit. If the program terminates normally, there will be a '.' in

TABLE 3: The syntax of ST

Syntax	Description
$Id ::= [a-zA-z\_]\,[a-zA-Z0-9\_]^*$ $Ids ::= Id^*$ $IdVal ::= Id := Expression$	Identifier
$EnumStructDeclaration ::= TYPE\ EnumDeclarationExp^*\ END\_TYPE$ $\mid TYPE\ StructDeclarationExp^*\ END\_TYPE$ $EnumBlock ::= Ids \mid IdVal^*$ $EnumDeclarationExp ::= Id : (EnumBlock)\ ; \mid Id : (EnumBlock) := Id;$ $StructDeclarationExp ::= Id : STRUCT\ VarDeclarationExp^*\ END\_STRUCT$	Enum and Struct declaration
$Function ::= FUNCTION\ Id\ :\ Type\ VarDeclaration^*\ Statements\ END\_FUNCTION$	Function declaration
$FunctionBlock ::= FUNCTION\_BLOCK\ Id\ VarDeclaration^*\ Statements\ END\_FUNCTION$	Function block declaration
$Program ::= PROGRAM\ Id\ VarDeclaration^*\ Statements\ END\_PROGRAM$	Program declaration
$Type ::= INT\|DINT\|SINT\|LINT\|UINT\|UDINT\|USINT\|ULINT\|BYTE\|WORD\|DWORD\|REAL$ $\|LREAL\|STRING\|STRING\,[Expression]\,\|WSTRING\|WSTRING\,[Expression]\,\|TIME\|DATE$ $\|TIME\_OF\_DAY\|DATE\_AND\_TIME\|Id\|ARRAY\,[Expression]\ OF\ Type$	Variable types
$VarType ::= VAR\_GLOBAL \mid VAR \mid VAR\_INPUT \mid VAR\_OUTPUT \mid VAR\_IN\_OUT \mid VAR\_TEMP$ $VarDeclarationExp ::= Ids : Type; \mid Ids : Type := Expression;$ $VarDeclaration ::= VarType\ \ VarDeclaration\ \ END\_VAR$	Variable declaration
$Operation ::= + \mid - \mid * \mid / \mid ** \mid MOD \mid < \mid > \mid = \mid <= \mid >= \mid <> \mid AND \mid \&$ $\mid AND\_THEN \mid XOR \mid OR \mid OR\_ELSE \mid ..$ $Expression ::= Int \mid Float \mid String \mid Bool \mid Bit \mid AllTime \mid Id \mid Expression\ Operation\ Expression$ $Expression\,(Expressions) \mid Expression.Expression \mid Expression\,[Expressions] \mid (Expression)$ $Expressions ::= Expression^*$	Expressions
$Assignment ::= Expression := Expression;$	Assignment statement
$ElseIfBlock ::= ELSE\ Statements \mid ELSE\_IF\ Expression\ THEN\ Statements\ ElseIfBlock^*$ $If ::= IF\ Expression\ THEN\ Statements\ ElseIfBlock^*\ END\_IF;$ $CaseBlock ::= Expression : Statements \mid Expression\ ..\ Expression : Statements$ $Case ::= CASE\ Expression\ OF\ CaseBlock^*\ END\_CASE;$ $\mid CASE\ Expression\ OF\ CaseBlock^*\ ELSE\ Statements\ END\_CASE;$	Branch statements
$While ::= WHILE\ Expression\ DO\ Statements\ END\_WHILE;$ $For ::= FOR\ Expression\ TO\ Expression\ DO\ Statements\ END\_FOR;$ $\mid FOR\ Expression\ TO\ Expression\ BY\ Expression\ DO\ Statements\ END\_FOR;$ $Repeat ::= REPEAT\ Statements\ UNTIL\ Expression\ END\_REPEAT;$	Loop statements
$Return ::= RETURN;$	Return statement
$Exit ::= EXIT;$	Exit statement
$Statement ::= Expression; \mid Assignment \mid If \mid Case \mid While \mid For \mid Repeat \mid Return \mid Exit$ $Statements ::= Statement^*$	Statements

the k cell, denoting that no more units need to be executed. In the preprocessing phase (the first pass of $\mathbb{K}$), the k cell only contains the token $execute$. Afterwards, $\mathbb{K}$ will start executing from the `MAIN` program.

Stack operations. The cell $control$ contains seven subcells—$fstack$, env, $temp$, $count$, $gvid$, $print$ and $break$—which record the operating environment of the currently running code segment. Specifically, the function stack $fstack$ is a list used to store the environment before executing other POUs, including variables in the current environment and the subsequent program. Next, the cell env is used to store the mapping relationship between variables and indexes in the current environment during program execution. Furthermore, cells $temp$ and $count$ are used in `ENUM` and `STRUCT`, where $temp$ is for temporary mapping and $count$ is used as a counting pointer. The cell $gvid$ records all identifiers of global variables to assist in the generation of global variables. The cell $print$ records variables which need to be output. Finally, $break$ stores the program after the loop in order to support the implementation of the `EXIT` statement in `FOR`, `WHILE` and `REPEAT` loops.

Execution environment. The $allenv$ cell is used to cache the execution environment before function calls (for strict

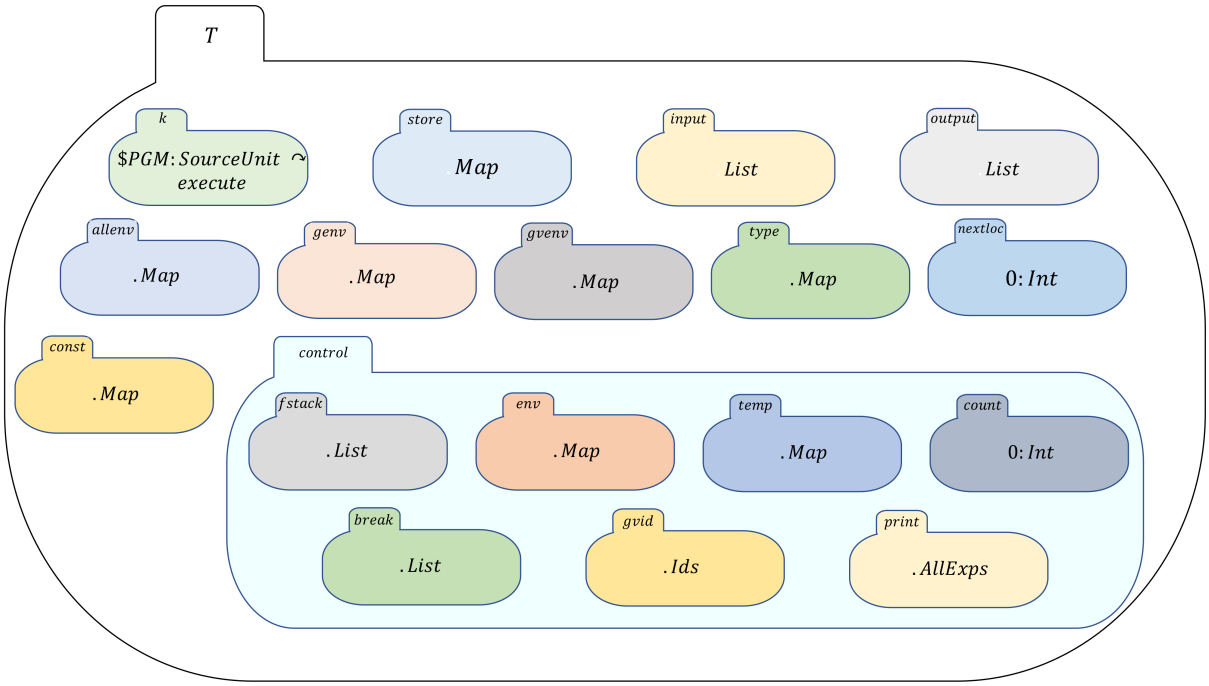

Fig. 5: The runtime configuration of ST in $\mathbb{K}$

type checking of parameter passing in function calls[1]). The cell $genv$ records the result of the pre-processing (including POUs and custom types) and will be copied to env when env is refreshed. The last cell related to the environment is called $gvenv$ and is used to index global variables.

Memory operation. The $store$ cell is used to simulate memory to record the mapping relationships of indexes and variable values. After that, the cells $input$ and $output$ are used to realize external inputs and external output respectively. The last cell, $nextLoc$, ensures that the index of a variable can always be incremented without duplication. The design consideration behind this is that for complex languages, it is more effective to explicitly manage arbitrarily large memory than use garbage collection [56].

3.3 Semantics of the Core Features

We implement the executable semantics covering most core features of ST and leave the vendor-specific functionalities as potential extensions. For example, some compilers would use additional keywords to distinguish the declaration part and the execution part of the program. In the following, we provide an overview of four core semantic features of ST, including 1) data types, 2) main control statements, 3) declarations and calls of POUs and 4) memory operations. Before diving into the details, we present the notations as follows.

3.3.1 Extended Data Types

The $\mathbb{K}$ framework supports diverse data types including identifiers (Id), integers (Int), bools ($Bool$), floats ($Float$) and strings ($String$), which cover most of the requirements. However, there are still some unsupported data types needing additional implementation in K-ST, which we call extended data types. These extended data types can be categorized into two kinds: 1) elementary types (`TIME`, `BYTE`, `WORD`, `DWORD`, `TIME_OF_DAY`, `DATE` and `DATE_AND_TIME`) and 2) compound types (`ENUM` and `STRUCT`). We implement these extended data types by the composition of built-in types and methods in $\mathbb{K}$ as follows.

We take `TIME_OF_DAY` as an example to introduce elementary types. There are two types of `TIME_OF_DAY` in ST, e.g., `TIME_OF_DAY#23:45:56.30` and `TOD#23:45:56.30`. Fig. 6 shows our implementation of `TIME_OF_DAY` type together with its relevant operations. Lines 1 and 2 respectively define the syntax of `TIME_OF_DAY` and how to parse it (`Get_TIME_OF_DAY`). Line 3 is used to convert `Get_TIME_OF_DAY` to `TIME_OF_DAY`, which is achieved by two steps—$Gtd2Td$ and $Standardization$—where $Gtd2Td$ realizes the conversion of the format and $Standardization$ realizes content conversion, e.g., replacing 60 minutes with 1 hour. Lines 4–11 define some arithmetic and relational operations of `TIME_OF_DAY`.

For compound types, we take `STRUCT` as an example and show its semantics in Fig. 7, including both `STRUCT` declaration and instantiation. Declarations are shown in **rule Struct_Declaration**, where we allocate memory for each defined data structure. The instantiation of `STRUCT` consists of four main steps in **rule Struct_Instantiation**: 1) $CreatStruct$ allocates memory for $I1$, 2) $StructInits$ generates each variable in turn according to Vds in `STRUCT`, 3) Set assigns values to the corresponding variables according to $Idvs$, and finally, 4) $Update$ stores the mapping relationship of variables related to $I1$ into the memory of $I1$ to facilitate subsequent use.

1. This is optional but recommended for ST compilers.

```
1  syntax TIME_OF_DAY ::= "TOD#" Int ":" Int ":" Int "." Int
2  syntax Get_TIME_OF_DAY ::= r"(TOD[\\#](([0-1]?[0-9])|([2][0-4]))[:][0-5]?[0-9][:][0-
   5]?[0-9]([.][0-9]{1,3})?)|(TIME_OF_DAY[\\#](([0-1]?[0-9])|([2][0-4]))[:][0-5]?[0-
   9][:][0-5]?[0-9]([.][0-9]{1,3})?)"
   [token]
3  rule GTD:Get_TIME_OF_DAY => Standardization(Gtd2Td(GTD))          [anywhere]
4  rule Td:TIME_OF_DAY + T:Time => Td +Tdt T
5  rule Td1:TIME_OF_DAY - Td2:TIME_OF_DAY => Td1 -Tdtd Td2
6  rule Td1:TIME_OF_DAY > Td2:TIME_OF_DAY => Td1 >Tdtd Td2
7  rule Td1:TIME_OF_DAY < Td2:TIME_OF_DAY => Td1 <Tdtd Td2
8  rule Td1:TIME_OF_DAY = Td2:TIME_OF_DAY => Td1 =Tdtd Td2
9  rule Td1:TIME_OF_DAY <> Td2:TIME_OF_DAY => Td1 <>Tdtd Td2
10 rule Td1:TIME_OF_DAY >= Td2:TIME_OF_DAY => Td1 >=Tdtd Td2
11 rule Td1:TIME_OF_DAY <= Td2:TIME_OF_DAY => Td1 <=Tdtd Td2
```

Fig. 6: Implementation of `TIME_OF_DAY` in $\mathbb{K}$

rule Struct_Declaration

$$\left\langle \frac{\text{TYPE } I : Id : \text{STRUCT } Vds : VarDeclaration\text{-}Exps \text{ END\_STRUCT END\_TYPE}}{.} \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [I \leftarrow L]} \right\rangle_{env} \left\langle \cdots \frac{.Map}{L \leftarrow Vds} \cdots \right\rangle_{store}$$
$$\left\langle \cdots \frac{.Map}{L \leftarrow struct} \cdots \right\rangle_{type} \left\langle \cdots \frac{.Map}{L \leftarrow false} \cdots \right\rangle_{constant}$$
$$\left\langle \cdots \frac{L}{L + Int\ 1} \cdots \right\rangle_{nextLoc}$$

rule Struct_Instantiation

$$\left\langle \frac{\text{VAR } I1 : Id : I2 : Id := (I : IdValues) \text{ END\_VAR}}{CreatStruct\,(I1) \curvearrowright StructInits\,(Vds,\ I1) \curvearrowright Set\,(I1,\ I) \curvearrowright Update\,(I1)} \cdots \right\rangle_{k}$$
$$\langle \cdots I2 \mapsto L \cdots \rangle_{env}$$
$$\langle \cdots L \mapsto Vds : VarDeclarationExps \cdots \rangle_{store}$$

Fig. 7: The partial semantics of `STRUCT` in $\mathbb{K}$

3.3.2 Main Control Statements

Control statements are important in ST for achieving complex program logic (as in most other programming languages). We show the rules for `CASE`, `REPEAT` and `EXIT` in Fig. 8 (as the semantics of `IF`, `WHILE` and `FOR` are typical). A `CASE` statement can be rewritten as a combination of an `IF` and `CASE` through **rule Case**. The **rule Repeat** is implemented as follows. We first store the subsequent statements outside the loop (recorded as K) in cell $break$ to deal with the `EXIT` statement that may appear, and then rewrite it into the form of `WHILE` for further execution. During the execution of the loop body, once `EXIT` is executed, all the statements in the current cell k are discarded and rewritten to K (storing the subsequent statements), as shown in **rule Exit**.

3.3.3 The Declaration and Call of POUs

In ST programs, statements are inside Program Organization Units (POUs), i.e., `FUNCTION`, `FUNCTION_BLOCK` or `PROGRAM`. A `FUNCTION` is a stateless POU type, comparing to a `FUNCTION_BLOCK` which stores its own state after execution. The design of the `FUNCTION_BLOCK` is similar to

rule Case

$$\left\langle \frac{\begin{array}{c}\text{CASE } E1 : Expression \text{ OF } E2 : Expression,\\ Es : Expressions\ :\ S : Statements\\ CB : CaseBlock \text{ END\_CASE;}\end{array}}{\begin{array}{c}\text{IF } E1 = E2 \text{ THEN } S \text{ ELSE}\\ \text{CASE } E1 \text{ OF } Es\ :\ S\ CB \text{ END\_CASE; END\_IF;}\end{array}} \cdots \right\rangle_{k}$$

rule Repeat

$$\left\langle \frac{\begin{array}{c}\text{REPEAT } S : Statements \text{ UNTIL}\\ E : Expression \text{ END\_REPEAT; } \curvearrowright K\end{array}}{S \curvearrowright \text{WHILE } (\text{NOT } E) \text{ DO } S \text{ END\_WHILE;}} \right\rangle_{k}$$
$$\left\langle \frac{.List}{ListItem\,(K)} \cdots \right\rangle_{break}$$

rule Exit

$$\left\langle \frac{\text{EXIT; } \curvearrowright \_}{K} \right\rangle_{k} \quad \left\langle \frac{ListItem\,(K)}{.List} \cdots \right\rangle_{break}$$

Fig. 8: The partial semantics of ST control statements

the concept of class-object manifestation in object-oriented programming (OOP), which aims to achieve better modularization. `FUNCTION_BLOCK`s exist in two forms: as a type or as an instance, and only the instance can be called. For a `FUNCTION_BLOCK` instance, the local variables retain their values between each 'call'. `PROGRAM`s are defined by the IEC 61131-3 standard as a "logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a PLC-system" [46]. Due to space constraints, we show the declaration, call and return operation of `FUNCTION_BLOCK`s in Fig. 9 as an example for illustration (`FUNCTION` and `PROGRAM` are shown in Fig. 10 and explained only when necessary).

Declaration. The declaration of `FUNCTION_BLOCK` is similar to that of `STRUCT`. As shown in **rule Function_Block_Declaration**, we first assign an index in memory for `FUNCTION_BLOCK` X, set the $type$ to the built-in $FunctionBlock$, and convert the entire declaration statement to the built-in type $funblambda(X, void, Vds, S)$ for storage, where $void$ means no return value, Vds and S are variable declarations and operations in X respectively. The purpose of setting $const$ to $true$ is to prevent it from being modified. Note that `FUNCTION` and `PROGRAM` set $type$ and $store$ to $Function$, $funblambda(X, T, Vds, S)$ and $Program$, $plambda(X, void, Vds, S, .Map)$.

Instantiation. The instantiation of `FUNCTION_BLOCK` is achieved through variable declarations, as shown in **rule Function_Block_Instantiation**. However, the value is set to $runfunblambda(X, void, Vds, S, .Map)$ to distinguish it from $funblambda$ and $.Map$ is designed to store the `FUNCTION_BLOCK` environment for next call and external query. This is because a `FUNCTION_BLOCK` can only be called after instantiation, i.e., $runfunblambda$ can be executed but $funblambda$ can not. Since `FUNCTION` and `PROGRAM` have no such restrictions, $funlambda$ and $plambda$ can be directly called and executed.

Call. There are two cases when a `FUNCTION_BLOCK`

rule Function_Block_Declaration

$$\left\langle \frac{\text{FUNCTION\_BLOCK } X : Id\ Vds : VarDeclar-\ ations\ S : Statements\ \text{END\_FUNCTION\_BLOCK}}{.} \ \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [X\ \leftarrow\ L]} \right\rangle_{env} \left\langle \cdots\ \frac{.Map}{L\ \mapsto\ FunctionBlock}\ \cdots \right\rangle_{type}$$
$$\left\langle \cdots\ \frac{.Map}{L \mapsto funblambda\,(X, void, Vds, S)}\ \cdots \right\rangle_{store}$$
$$\left\langle \cdots\ \frac{.Map}{L\ \mapsto\ true}\ \cdots \right\rangle_{constant} \langle L\ \Rightarrow\ L\ +Int\ 1\rangle_{nextLoc}$$
$$requires\ notBool\ X\ in\_keys\,(Env)$$

rule Function_Block_Instantiation

$$\left\langle \frac{\text{VAR } X1\ :\ funblambda\,(X2, void, Vds, S)\ \text{END\_VAR}}{.} \ \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [X1\ \leftarrow\ L]} \right\rangle_{env} \left\langle \cdots\ \frac{.Map}{L\ \mapsto\ X2}\ \cdots \right\rangle_{type}$$
$$\left\langle \cdots\ \frac{.Map}{L \mapsto runfunblambda\,(X1, void, Vds, S, .Map)}\ \cdots \right\rangle_{store}$$
$$\left\langle \cdots\ \frac{.Map}{L\ \mapsto\ true}\ \cdots \right\rangle_{constant} \langle L\ \Rightarrow\ L\ +Int\ 1\rangle_{nextLoc}$$
$$requires\ notBool\ X\ in\_keys\,(Env)$$

rule Function_Block_Call_First

$$\left\langle \frac{runfunblambda(X,\ T,\ Vds : VarDeclarations,\ S : Statements,\ .Map)\,(E : Expressions)\ \curvearrowright\ K}{renew\ \curvearrowright\ runpfblambda\,(X, T, Vds, S, .Map)\,(E)\ \curvearrowright Update\,(X)\ \curvearrowright RETURN\ null;} \right\rangle_{k}$$
$$\left\langle \left\langle \frac{.List}{ListItem(info(X,T,K,Allenv,C))} \right\rangle_{fstack} \left\langle \frac{Env}{Genv} \right\rangle_{env} C \right\rangle_{control}$$
$$\langle Genv \rangle_{genv} \left\langle \frac{Allenv}{Env} \right\rangle_{allenv}$$

rule Function_Block_Call_Others

$$\left\langle \frac{runfunblambda(X : Id,\ T : Type,\ Vds,\ S,\ M)\ (E : Expressions)\ \curvearrowright\ K}{renew\ \curvearrowright\ runpfblambda\,(X, T, Vds, S, M)\,(E)\ \curvearrowright Update\,(X)\ \curvearrowright RETURN\ null;} \right\rangle_{k}$$
$$\left\langle \left\langle \frac{.List}{ListItem(info(X,T,K,Allenv,C))} \right\rangle_{fstack} \left\langle \frac{Env}{M} \right\rangle_{env} C \right\rangle_{control}$$
$$\left\langle \frac{Allenv}{Env} \right\rangle_{allenv} \langle Genv \rangle_{genv} \quad requires\ M\ \neq K\ .Map$$

rule Return

$$\left\langle \frac{\text{RETURN } V : Value;\ \curvearrowright\ \_}{V\ \curvearrowright\ K} \right\rangle_{k}$$
$$\left\langle \left\langle \frac{ListItem(info(\_,\_,K,\ M,\ C))}{.List} \right\rangle_{fstack} \left\langle \frac{\_}{Allenv} \right\rangle_{env} (\_ \Rightarrow C) \right\rangle_{control}$$
$$\left\langle \frac{Allenv}{M} \right\rangle_{allenv}$$

Fig. 9: The partial semantics of `FUNCTION_BLOCK`

is called. The first case is that the `FUNCTION_BLOCK` is called for the first time, as shown in **rule Function_Block_Call_First**. Since there is no initial environment (the last value of $runfunblambda$ is $.Map$), we will first store the current execution environment $info$ in $fstack$, including subsequent statements K, the $Allenv$ of the current environment, and the parameters C in cell $control$. Then, we reset parameters C through $renew$. After that, $\mathbb{K}$ executes the variable declaration Vds (including index application, initialization and assignment) and statements S in the function block. In addition, $Update$ is used to update the $.Map$ in $runfunblambda$ to record the current environment. Finally, `RETURN` can return to the calling program and configure the corresponding environment. In other cases (not called for the first time), as shown in **rule Function_Block_Call_Others**, there is already a mapping relationship between related variables and values in cell $store$, and the mapping relationship between identifiers and indexes is also stored in the $runfunblambda$. Therefore, no new memory allocation will be made during the execution process and the existing environment will be used. Note that the value of the variable in the `FUNCTION_BLOCK` will not be initialized, which means that the execution result for the same input may be different.

Regardless of whether `RETURN` appears in the `FUNCTION_BLOCK`, we add a `RETURN` by default for each `FUNCTION_BLOCK` as a sign that the `FUNCTION_BLOCK` has finished running and returned to the calling POUs. Since `FUNCTION_BLOCK`s and `PROGRAM`s do not have a return value, we set $null$ as the return value. Note that a `FUNCTION` has a return value, and the returned value is the value corresponding to the function identifier, so we need to use $Clearenv$ to clean up the memory environment corresponding to the function identifier after calling procedure $renew$ and add the declaration of the function identifier variable in Vds.

3.3.4 Memory Operations

Here, we present the rules for memory operations on elementary types in ST, such as built-in types and extended elementary types. What elementary types have in common is that they take only one memory slot. For complex types, such as enums, structs, arrays, etc, which are compositions of elementary types, the memory operation can be regarded as a set of memory operations on elementary types. For instance, the assignment to struct can be equivalent to assign value for each variable of this struct.

Similar to ST_{demo}, main memory operations in ST are still composed of $Allocation$, $Lookup$, $Assignment$ and additional $Clearenv$. Where $Allocation$ implements the allocation of memory for variables in the $store$, $Lookup$ is used to find variable values in $store$ cell, $Assignment$ implements the assignment of variables, and $Clearenv$ implements the recovery of memory in the $store$. However, because the complete ST semantics has a more complex type design, they will involve more cells in configurations, and are more complicated, as shown in Fig. 11.

Note that $HOLE$ is just a variable, but it has special meaning in the context of sentences with the 'heat' or 'cool' attribute. In short, 'heat' is to lookup the corresponding content of the $HOLE$ in the formula, and 'cool' is to put the recheck results back into the formula. For example, in expression $a+b$ where a is represented as $HOLE$, 'heat' is to take a out of the formula and find its corresponding value. If it is 3, and 'cool' puts 3 back into the original formula, then the formula becomes $3 + b$.

Let us start with the $Assignment$ operation (we omit $Lookup$ as it is straightforward). The $Assignment$ of ST

rule Function_Declaration

$$\left\langle \frac{\text{FUNCTION } X:Id\ :\ T:Type\ Vds:VarDeclarations\ S:Statements\ \text{END\_FUNCTION}}{.} \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [X\ \leftarrow\ L]} \right\rangle_{env} \quad \left\langle \cdots \frac{.Map}{L\ \mapsto\ Function} \cdots \right\rangle_{type}$$
$$\left\langle \cdots \frac{.Map}{L\ \mapsto funlambda\,(X,T,Vds,S)} \cdots \right\rangle_{store}$$
$$\left\langle \cdots \frac{.Map}{L\ \mapsto\ true} \cdots \right\rangle_{constant} \quad \langle L\ \Rightarrow\ L\ +Int\ 1\rangle_{nextLoc}$$
$$requires\ notBool\ X\ in\_keys\,(Env)$$

rule Function_Call

$$\left\langle \frac{funlambda(X:Id,\ T:Type,\ Vds:VarDeclarations,\ S:Statements)\,(E:Expressions)\ \curvearrowright\ K}{renew\ \curvearrowright\ ClearEnv\,(X)\ \curvearrowright runpfblambda\ (X,T,\text{VAR } X\ :\ T;\ \text{END\_VAR } Vds,S,.Map)\,(E)\ \curvearrowright RETURN\ X;} \right\rangle_{k}$$
$$\left\langle \left\langle \frac{.List}{ListItem(info(X,T,K,Allenv,C))} \right\rangle_{fstack} \left\langle \frac{Env}{Genv} \right\rangle_{env} C \right\rangle_{control}$$
$$\left\langle \frac{Allenv}{Env} \right\rangle_{allenv} \langle Genv\rangle_{genv}$$

rule Program_Declaration

$$\left\langle \frac{\text{PROGRAM } X:Id\ Vds:VarDeclarations\ S:Statements\ \text{END\_PROGRAM}}{.} \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [X\ \leftarrow\ L]} \right\rangle_{env} \quad \left\langle \cdots \frac{.Map}{L\ \mapsto\ Program} \cdots \right\rangle_{type}$$
$$\left\langle \cdots \frac{.Mao}{L\ \mapsto plambda\,(X,void,Vds,S,.Map)} \cdots \right\rangle_{store}$$
$$\left\langle \cdots \frac{.Map}{L\ \mapsto\ true} \cdots \right\rangle_{constant} \quad \langle L\ \Rightarrow\ L\ +Int\ 1\rangle_{nextLoc}$$
$$requires\ notBool\ X\ in\_keys\,(Env)$$

rule Program_Call

$$\left\langle \frac{plambda(X,T,Vds,S,M)\,(E)\ \curvearrowright\ K}{renew\ \curvearrowright\ runpfblambda\,(X,T,Vds,S,M)\,(E)\ \curvearrowright Update\,(X)} \right\rangle_{k}$$
$$\left\langle \left\langle \frac{.List}{ListItem(info(X,T,K,Allenv,C))} \right\rangle_{fstack} \left\langle \frac{Env}{Genv} \right\rangle_{env} C \right\rangle_{control}$$
$$\left\langle \frac{Allenv}{Env} \right\rangle_{allenv} \langle Genv\rangle_{genv}$$

Fig. 10: The partial semantics of `FUNCTION` and `PROGRAM`

rule Lookup

$$\left\langle \frac{X\ :\ Id}{V} \cdots \right\rangle_{k} \langle \cdots X\ \mapsto\ L\ \cdots\rangle_{env}$$
$$\langle \cdots L\ \mapsto\ V\ :\ Value\ \cdots\rangle_{store}$$
context $(HOLE\ \Rightarrow\ lvalue\,(HOLE)) := \_$

rule Find_Index

$$\langle lvalue\,(X\ :\ Id\ \Rightarrow\ loc\,(L))\ \cdots\rangle_{k} \quad \langle \cdots X\ \mapsto\ L\ :\ Int\ \cdots\rangle_{env}$$

rule Assignment

$$\left\langle \frac{loc\,(L\ :\ Int)\ :=\ V\ :\ Value}{V} \cdots \right\rangle_{k}$$
$$\langle \cdots L\ \mapsto\ (\_\ \Rightarrow\ V)\ \cdots\rangle_{store} \quad \langle \cdots L\ \mapsto\ false\ \cdots\rangle_{constant}$$
$$\langle \cdots L\ \mapsto\ T\ :\ EleType\ \cdots\rangle_{type} \quad requires\ Limit(V,\ T)$$

rule Allocation

$$\left\langle \frac{VAR\ X\ :\ Id\ :\ T\ :\ EleType;\ END\_VAR}{.} \cdots \right\rangle_{k}$$
$$\left\langle \frac{Env}{Env\ [X\ \leftarrow\ L]} \right\rangle_{env} \quad \langle Gvenv\rangle_{gvenv}$$
$$\left\langle \cdots \frac{.Map}{L\ \mapsto\ Undefined\,(T)} \cdots \right\rangle_{store} \quad \left\langle \cdots \frac{.Map}{L\ \mapsto\ T} \cdots \right\rangle_{type}$$
$$\left\langle \cdots \frac{.Map}{L\ \mapsto\ false} \cdots \right\rangle_{constant} \quad \langle L\ \Rightarrow\ L\ +Int\ 1\rangle_{nextLoc}$$
$$requires\ notBool\ (X\ in\_keys\ \ (Env)\ \ orBool\ X\ in\_keys\ \ (Gvenv))$$

rule Clearenv

$$\left\langle \frac{Clearenv\,(X\ :\ Id)}{.} \cdots \right\rangle_{k} \quad \left\langle \frac{Env}{Env\ [X\ \leftarrow\ undef]} \right\rangle_{env}$$

Fig. 11: The partial semantics of memory operations

divides the $Assignment$ of ST_{demo} into two steps, where **context** and **rule Find_Index** are used to determine the index L of the assigned variable X in $store$, and **rule Assignment** implements the update of the $store$ at index L. The purpose of this division is to make the $Assignment$ operation better applicable to complex types, because in some cases the index of the assigned variable can not be directly obtained and multiple queries are required. For instance, when assigning a value to $A\,[3,5,7]$, where A is a multi-dimensional array, we need to look up each dimension one by one to finally determine the index. In addition, we refer to the state of X in $type$ and $constant$ during the assignment process. On the one hand, we use $Limit$ to ensure that the assigned value meets the type requirements, and on the other hand, we use $constant$ to ensure that the constant cannot be modified. Although the memory cleaning operation is not necessary for ST in $\mathbb{K}$, a simple $Clearenv$ operation can effectively reduce repetitive code and improve code readability. For **rule Clearenv**, what needs attention is the operation on cell env: it replaces the index L of variable X with $undef$ which means null in the map supported by $\mathbb{K}$.

ST has relatively complex and strict type definitions, therefore the **rule Allocation** of ST involves more cells and operations, such as $type$ and $constant$ for storing variable types and whether they are constants, where $Undefined$ is used to generate the default of the specified type. In addition, according to the content in TABLE 1, not only `VAR` will be used in the variable declaration process, but also other keywords, such as `VAR_INPUT`, `VAR_IN_OUT`, etc. In order to reduce the complexity of the code, we also implement these declarations through `VAR` declarations. For instance, Fig. 12 shows the implementation of `VAR_GLOBAL` and `CONSTANT`. We realize regional changes (from the env cell to the $gvenv$ cell) through $letogv$, and $SetConstant$ realizes the modification of the value in the $const$ cell.

We remark that K-ST covers 259 core features with 876 rules in total, using 2315 lines of $\mathbb{K}$ code. The complete code can be accessed through https://github.com/wkyml/K-ST.

rule Global_Declaration

$$\left\langle \frac{\text{VAR\_GLOBAL } X : Id \ : \ T : EleType; \text{ END\_VAR}}{\text{VAR } X \ : \ T; \text{ END\_VAR } \curvearrowright \ letogv\,(X) \ \curvearrowright \ ClearEnv\,(X)} \ \cdots \right\rangle_k$$

rule Constant_Declaration

$$\left\langle \frac{\text{VAR CONSTANT } X : Id \ : \ T : EleType; \text{ END\_VAR}}{\text{VAR } X \ : \ T; \text{ END\_VAR } \curvearrowright \ SetConstant\,(X)} \ \cdots \right\rangle_k$$

Fig. 12: The partial semantics of variable declarations

4 Testing and Analysing ST Compilers

In addition to providing formal references for defined languages, our formal semantics also has several applications that use language-independent tools provided by $\mathbb{K}$, such as state space exploration, model checking, symbol execution and deductive program validation. We omit demonstration of these applications in this paper since they have been well-illustrated in related works [51], [52]. In this work, we introduce the testing of ST implementations/compilers based on our executable semantics, K-ST.

As discussed earlier, because ST compilers are typically provided by vendors, the execution behavior of compilers may be different, and may even be inconsistent with respect to the high-level semantics [37]. One of the main applications of the proposed semantics is to define the 'reference' execution behavior of ST, which can help programmers detect bugs in existing ST compilers.

To explore this application (and given the closed nature of commercial compilers), we choose OpenPLC[2] as our test object, which is open source and supports ST programming. The overall workflow of our testing approach is depicted in Fig. 13. It includes three parts: program variation, program execution and result comparison. First, seed programs are mutated to improve the diversity of test samples. Next, we use the mutated program as input to run OpenPLC and our executable semantics respectively. Finally, the result comparison part compares the consistency of the two execution results. It should be noted that we use a policy similar to [33], that is, the program does not need input, and the category of result consistency comparison includes the values of all variables in the program. The comparison of results is performed to analyze potential inconsistencies between K-ST and OpenPLC. By comparing the final execution state of the program with its variable state, we can identify potential inconsistencies. The execution state focuses on determining whether the program has completed its execution or terminates at the same statement. On the other hand, the variable state captures the values of all variables in the program, including input, output, and intermediate variables, after the program has finished running. TABLE 4 shows our measure of consistency, where Q and Q' represent the values of each variable after the program executes, I and I' represent the commands corresponding to the exception termination, and ✔and ✗represent consistency and inconsistency respectively. As a result, unless K-ST and OpenPLC exhibit identical execution and memory states, their behavior will be deemed inconsistent.

2. https://www.openplcproject.com/

TABLE 4: Measure of K-ST/OpenPLC consistency

		The result of K-ST	
		Successful execution(Q)	Unusual termination(I)
The result of OpenPLC	Successful execution(Q')	$Q = Q'$ ✔ $Q \neq Q'$ ✗	✗
	Unusual termination(I')	✗	$I = I' \& Q = Q'$ ✔ $others$ ✗

TABLE 5: Mutation operations

Mutation Operation	Example
Variable Random Assignment	$a : INT; \rightsquigarrow a : INT := 3527;$
Scalar Variable Replacement	$a := b; \rightsquigarrow a := c \mid 30;$
Arithmetic Operator Replacement	$a + b \rightsquigarrow a - b$
Arithmetic Operator Insertion	$a + b \rightsquigarrow a + b - c$
Arithmetic Operator Deletion	$a + b - c \rightsquigarrow a + b$
Relational Operator Replacement	$a > b \rightsquigarrow a <= b$
Logical Connector Replacement	$a \ AND \ b \rightsquigarrow a \ OR \ b$
Logical Connector Insertion	$a \ AND \ b \rightsquigarrow a \ AND \ b \ OR \ c$
Logical Connector Deletion	$a \ AND \ b \ OR \ c \rightsquigarrow a \ AND \ b$
"NOT" Mutation	$NOT \ a \rightsquigarrow a \mid a \rightsquigarrow NOT \ a$
Statement Insertion	$\rightsquigarrow IF \cdots END\_IF;$
Statement Deletion	$EXIT; \rightsquigarrow$

In order to better mutate seed programs to improve the diversity of test samples, we propose specific mutation operations in TABLE 5 to generate mutated test samples. These mutation operations can enrich the test samples while minimizing program errors. Our method for generating mutant ST programs is shown in Algorithm 1. Given an ST program S_i, the algorithm makes a copy, randomly assigns initial values to all variables at the time of declaration, and applies some applicable mutation operators to randomly selected lines in the program. The test is done by comparing results of these samples in K-ST and OpenPLC. It should be noted that correct and erroneous programs in the test sample are both meaningful for checking the consistency of execution behavior. This is because K-ST and OpenPLC report program errors at the same time, allowing us to verify a stronger notion of consistency. In addition, considering the lag of OpenPLC updates, we also tested it on the latest Beremiz[3] which uses the same underlying implementation (MATIEC [4]) as OpenPLC. The specific results of the test are shown in Section 5.

5 Evaluation

In order to evaluate the semantics of ST we defined in $\mathbb{K}$, we deployed K-ST on $\mathbb{K}$ version 5.1.11 (Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz). In the following, we design multiple experiments to systematically answer the following research questions (*RQs*).

3. https://beremiz.org/
4. https://github.com/thiagoralves/OpenPLC_Editor/tree/master/matiec

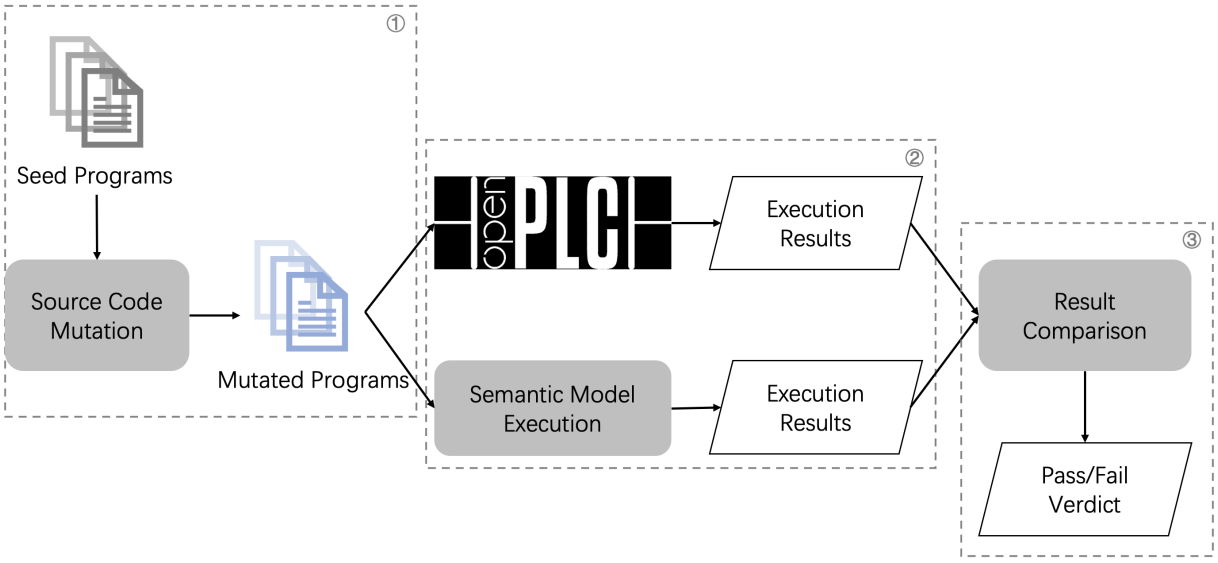

Fig. 13: Overview of the test process

Algorithm 1: Generating mutated ST programs

Input: A set of ST programs S
Output: A set of mutation ST programs S_M

Let Op_{vra} be the Variable Random Assignment;
Let Ops be the other mutation operations;
Let $S_M := \emptyset$;
for $S_i \in S$ **do**
  Make a copy $S_i^{'}$ of the ST program S_i;
  $mutated := false$;
  Apply Op_{vra} to all variable declarations;
  while $mutated$ **do**
    Randomly choose a set of lines number I from $S_i^{'}$;
    for $i \in I$ **do**
      if *randomly choose operator* $Op \in Ops$ *is applicable to line* i **then**
        Apply Op to line i;
        $mutated := true$;
  $S_M := S_M \cup S_i^{'}$;
return S_M;

- *RQ1: How much of the ST language is K-ST covering?* Completeness of the semantics is an important indicator to measure executable formal semantics. The lack of key semantics will seriously affect the usefulness of formal semantics.
- *RQ2: Is K-ST correct?* Semantic correctness is the basis for ensuring the usability of executable formal semantics, so we need to analyze the correctness of formal semantics implemented.
- *RQ3: Can K-ST be used to discover bugs in a compiler?* This is important since a key application of executable formal semantics is to identify compiler bugs.

5.1 Test Sets

For the purpose of evaluating the coverage and the correctness of K-ST, the test data set that we used comes from GitHub. We searched 4853 programs in GitHub through keywords in the ST language. Then, we automatically screened out samples containing other programming languages (2516) and XML forms (1542). After that, we manually splice the remaining programs and remove samples that lack the components required for operation (such as POUs). After screening, 567 complete programs written in pure ST formed our test set. In other words, these 567 samples contain all the components required for operation and do not use other languages, such as C and Python.

With the aim of comprehensively testing the correctness of the execution behavior of OpenPLC, we use two sample sets, including test samples collected from GitHub (GitHub set) and test samples obtained through mutation (Mutated set). The GitHub set is the sample set with 567 test samples mentioned before. The Mutated set is generated by Algorithm 1. We selected 30 high-quality samples from GitHub set as initial mutant seeds. These 30 samples contain all the key features of ST. Then, three rounds of iterative mutation are carried out through Algorithm 1. Each round of iteration produces 10 mutation samples per seed. Except for the initial seed used in the first round, the seeds of each round of mutation are the result of the previous round of mutation. We get a set containing 33,330 mutation samples.

5.2 Experiment Results and Analyses

5.2.1 Semantic Completeness (RQ1)

We executed K-ST on 567 test samples collected from GitHub. Among these 567 test samples, K-ST supports the execution of 509 of them. For these 509 tests which K-ST can support, Fig. 14 lists the number of tests for some important features (based on TABLE 1) used in the evaluation. Specifically, there are six kinds of features, namely `FUNCTION`, `FUNCTION_BLOCK`, `PROGRAM`, *Declaration types*, *Date types*

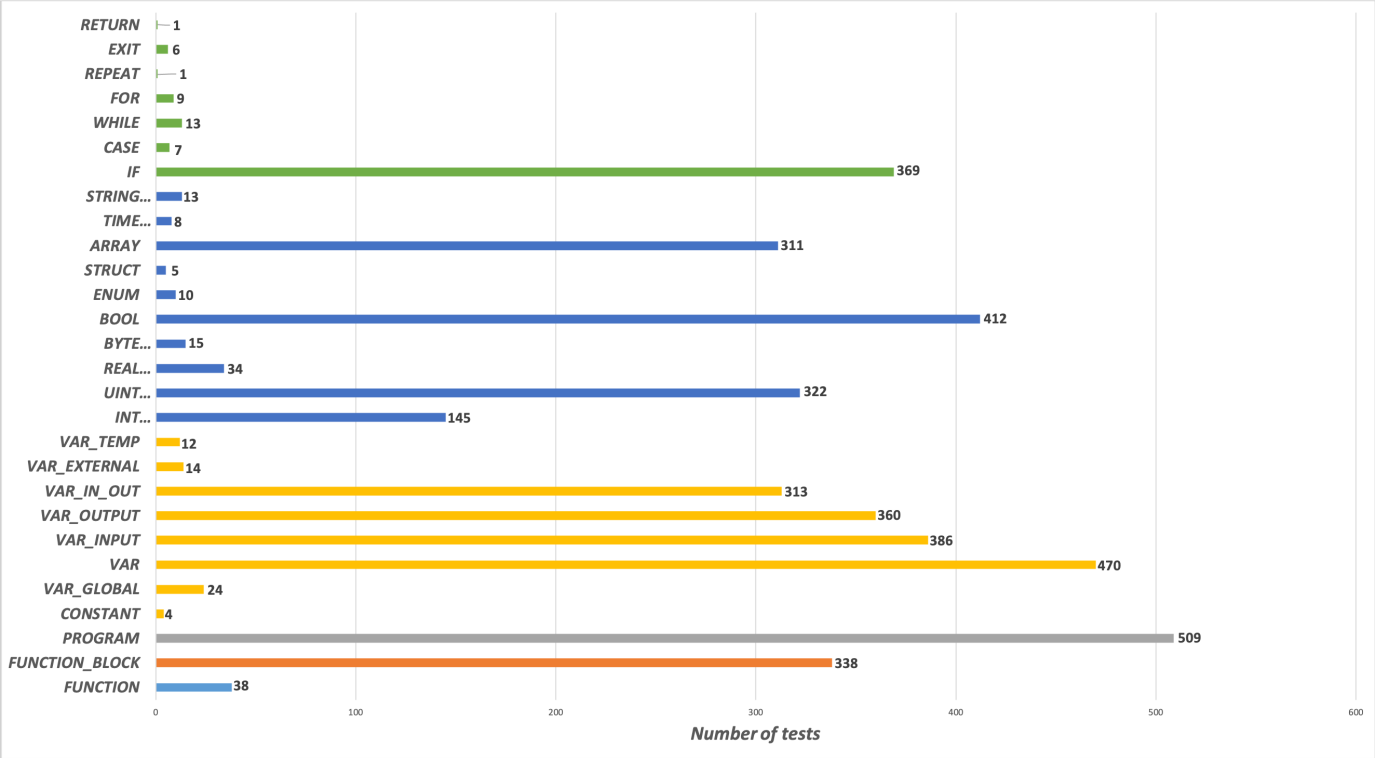

Fig. 14: Number of tests for each feature in ST

and *Statements*. For *Declaration types*, we list the number of tests for `CONSTANT`, `VAR_GLOBAL`, `VAR`, `VAR_INPUT`, `VAR_OUTPUT`, `VAR_IN_OUT`, `VAR_TEMP` and `VAR_EXTERNAL`. For *Data types*, we list the number of tests for elementary types signed integer (`INT`, `DINT`, `SINT`, `LINT`), unsigned integer (`UINT`, `UDINT`, `USINT`, `ULINT`), float (`REAL`, `LREAL`), Boolean (`BOOL`), byte (`BYTE`, `WORD`, `DWORD`), string (`STRING`, `WSTRING`), and time (`TIME`, `DATE`, `TIME_OF_DAY`, `DATE_AND_TIME`); compound types enum (`ENUM`) and struct (`STRUCT`); and finally, the array type `ARRAY`. For *Statements*, we list the number of tests for main control statements: `IF`, `CASE`, `FOR`, `WHILE`, `REPEAT`, `EXIT` and `RETURN`.

As indicated in Fig. 14, compared with `FUNCTION`, the `FUNCTION_BLOCK` is more favored by ST programmers (`PROGRAM` is necessary for ST program operation). For *Declaration types*, the most used is `VAR` (with a ratio of $470/509$), followed by `VAR_INPUT` ($386/509$), `VAR_OUTPUT` ($360/509$) and `VAR_IN_OUT` ($313/509$). Among all the *Data types*, `BOOL` is the most used, followed by unsigned integer and `ARRAY`, constituting $322/509$ and $311/509$ respectively. For the *Data types*, `BOOL` is the most common type. In addition, we must remark that we do not count the type of array members. Finally, `IF` is the most common statement in all the tests considered. This is also in line with the main working scenarios of PLCs.

We remark that we do not consider the vendor-based functions in this experiment as these functions vary not only from vendor to vendor, but even from product to product. In particular, Mitsubishi PLCs provide completely different data types, including Bit, Word[Signed/Unsigned], Double Word[Signed/Unsigned], Bit STRING[16-bit/32-bit], FLOAT, STRING[32] and Time. Siemens PLCs support keyword `BEGIN` to represent the end of variable declaration and the beginning of operation instructions. In addition, there are also obvious differences between different products of the same vendor. For example, the S7-1500 and the S7-1200 from Siemens support different type conversion methods[5], where the former only provides explicit conversions of types, and the latter provides both explicit and implicit conversions.

5.2.2 Semantics Correctness (RQ2)

On the other hand, in order to evaluate the correctness of K-ST, we compared the execution results of K-ST against those of vendor compilers CODESYS, CX-Programmer and GX Works2. We consider the proposed semantics correct if the execution behaviors of K-ST are consistent with the ones of the CODESYS, CX-Programmer and GX Works2 compilers. The consistency criteria described in Section 4 are utilized to evaluate the consistency of behavior between K-ST and the compilers provided by vendors. Specifically, if K-ST and these compilers demonstrate identical execution and variable states for the same program, their behavior is deemed consistent. We list the coverage of the K-ST semantics in TABLE 6 from the perspective of each feature specified by the official ST documentation, where FC, C and N mean "Fully Covered and Consistent with Compilers", "Covered and Consistent with Compilers" and "Not Covered", respectively.

From TABLE 6, we can see clearly that for POUs, we fully cover the declaration and call. In variable declarations,

5. https://support.industry.siemens.com/dl/dl-media/272/109742272/att_918238/v6/93516999691/zh-CHS/index.html#ae443583b99950f7cca0d7237fe81ad4

TABLE 6: Coverage of the proposed ST semantics

Feature	Coverage	Feature	Coverage	Feature	Coverage
POUs(core)		**Data types(core)**		*Enum instantiation*	FC
POUs declaration		*SINT*	FC	*Struct*	
FUNCTION	FC	*INT*	FC	*Struct declaration*	FC
FUNCTION_BLOCK	FC	*DINT*	FC	*Struct instantiation*	FC
PROGRAM	FC	*LINT*	FC	*Function block*	
POUs calls		*USINT*	FC	*Function block instantiation*	FC
FUNCTION	FC	*UINT*	FC	*Array*	
FUNCTION_BLOCK	FC	*UDINT*	FC	$One-dimensional\ array$	C
PROGRAM	FC	*ULINT*	FC	$Multi-dimensional\ array$	C
Variable Declaration(core)		*REAL*	FC	**Statements(core)**	
CONSTANT	FC	*LREAL*	FC	*Assignment statement*	
VAR_GLOBAL	FC	*BOOL*	FC	:=	FC
VAR	FC	*BYTE*	FC	$\Rightarrow$	N
VAR_INPUT	FC	*WORD*	FC	*Branch statement*	
VAR_OUTPUT	FC	*DWORD*	FC	*IF*	FC
VAR_IN_OUT	FC	*STRING*	FC	*CASE*	FC
VAR_EXTERNAL	FC	*WSTRING*	FC	*Loop statement*	
VAR_TEMP	FC	*TIME*	FC	*WHILE*	FC
AT	C	*DATE*	FC	*FOR*	FC
RETAIN	N	*TIME_OF_DAY*	FC	*REPEAT*	FC
PERSISTENT	N	*DATE_AND_TIME*	FC	*Break statement*	
Typed constant		*Enum*		*RETURN*	FC
Type # Data	FC	*Enum declaration*	FC	*EXIT*	FC

Built − in function

Numerical function (30)

ADD, SUB, MUL, SQR, INC, DEC, MAX, MIN, MUX, ABS, SQRT, TRUNC, FRAC, FLOOR, LN, LOG, EXP, SIN COS, TAN, COS, TAN ASIN, ACOS, ATAN, NEG, EXPT, DIV, MOD, LIMIT

Logical function (9)

GT, LT, GE, LE, EQ, NE, AND, OR, SEL

String function (9)

CONCAT, INSERT, DELETE, REPLACE, FIND, LEN, LEFT, RIGHT, MID

Translate function (160)

☐ FC: Fully Covered and Consistent with Compilers (256/262) ☐ C: Covered and Consistent with Compilers (3/262) ☐ N: Not Covered (3/262)

`AT` is related to input and output. We remark, however, that the storage mode of variables in $\mathbb{K}$ is very different from that in real PLCs, so we just support simple computer-side input and output. In addition, `RETAIN` and `PERSISTENT` are related to the actual situation in the PLC, so they are not implemented. For instance, `AT` is used to bind the actual point of the PLC; `RETAIN` and `PERSISTENT` support the preservation of variable values after a power failure or power loss. *Array* is the only one which is covered but not fully covered in all data types. Limited by the realization of arrays, it is temporarily impossible to achieve the array for enum and struct, and to assign values to multi-dimensional arrays as a whole. In statements, $\Rightarrow$ has been used in $\mathbb{K}$ and can be replaced by :=. For built-in functions, we show a list which we supported, including 30 numerical functions, 9 logical functions, 9 string functions and 160 translate functions.

In the process of comparing with CODESYS, CX-Programmer and GX Works2, the following points need to be explained. Firstly, due to the closed nature of these compilers, they cannot be simply called, so we have to manually fill the code in the specified way into the compiler to compile and run, and compare the results, which is laborious and tedious work. This also hinders us from testing these commercial compilers in an extensively large scale. After that, different vendors have obvious differences in the implementation of compilers, so the source code needs to be adapted to a certain extent. For example, only 10 basic data types—Bit, Word[Signed/Unsigned], Double Word[Signed/Unsigned], Bit STRING[16-bit/32-bit], FLOAT, STRING[32] and Time—are provided in the GX Works2 compiler, so we need to adapt the variable types of the source program.

TABLE 7: The results of K-ST and OpenPLC

Data Set		GitHub Set	Mutated Set
Number of samples		567	31059 (2271)
Number of program run completely	K-ST	509	15850
	OpenPLC	490	11581
Inconsistent	K_pO_f	30	5664
	K_fO_p	11	1395
	Diff. Result	0	735

5.2.3 Finding Bugs in OpenPLC (RQ3)

We execute OpenPLC and K-ST with the GitHub set and Mutated set as input. The execution results of the two data sets are shown in TABLE 7. Here, K_pO_f is the number of programs that K-ST can execute normally but OpenPLC cannot compile and run; K_fO_p is the number of programs that K-ST cannot run normally but OpenPLC can.

For the GitHub set, K-ST supports 509 of them, and OpenPLC supports 490. Through analysis, we found that the reason for this phenomenon is that OpenPLC has some functional deficiencies. For example, OpenPLC does not support the initialization of variables using formulas at the time of declaration; numerical calculations of `BYTE`, `WORD`, `DWORD` types are not supported, etc.

For the Mutated set, there is a big difference between the execution results of K-ST and OpenPLC. First of all, we filter 2,271 timeout programs that timed out both in OpenPLC and K-ST with 10 seconds as the time limit. After that, we manually analyzed these samples with inconsistent results to determine the causes. For the large K_pO_f value, functional deficiencies remain the main reason.

We found an interesting bug in OpenPLC. The bug is a "VAR" parsing exception in OpenPLC. If the first operation instruction starts with "VAR", such as "VAR0 := 1;", OpenPLC terminates abnormally. The interesting phenomenon is when an error statement appears in an unexecuted part of the program, such as after the "`RETURN;`": K-ST can execute such a program, but OpenPLC cannot. The main reason for this phenomenon is that $\mathbb{K}$ adopts an operation-based detection mechanism. Because the error code will not be executed, it will not lead to the termination of our executable semantics. The case study is shown in APPENDIX A.

After that, by analyzing those programs that have different results on K-ST and OpenPLC, we find that the reasons for the different results are mainly due to the differences in underlying implementations between $\mathbb{K}$ and OpenPLC. For example, for integer mode operation -7 MOD 3, the execution result of K-ST is -1, whereas the result for OpenPLC is 2. From a mathematical point of view, both results are correct, but they will have a completely different impact on any following operations. When we run the program again in CODESYS, the results of CODESYS were the same as K-ST.

For those samples that K-ST cannot run normally but OpenPLC can execute normally, our analysis found some bugs in OpenPLC. For example, while OpenPLC can check explicit divide-by-zero operations, it allows the execution of implicit divide-by-zero operations. TABLE 8 details all functional deficiencies and bugs we found in OpenPLC. We show some relevant case studies in APPENDIX B. Considering that Beremiz can be regarded as an updated version of OpenPLC, we have retested the inconsistencies we found in Beremiz. We found that in the latest Beremiz, it fixes some problems, including negative MOD operation results and "VAR" parsing exceptions. But other bugs and shortcomings still exist. In response to these problems in OpenPLC, we have submitted them to the OpenPLC and Beremiz developers and are waiting for their confirmation[6].

6 Related Work

In this section, we discuss some other PLC program analysis techniques, summarize their characteristics, and distinguish them from our work.

Keliris et al. [19] propose a framework (ICSREF) which can automate the reverse engineering process for PLC binaries. They instantiate ICSREF modules for reversing binaries compiled with CODESYS and getting the complete Control Flow Graph (CFG), and they provide an end-to-end case study of dynamic payload generation and attack deployment. Tychalas et al. [7] analyze the binary files generated by all control system programming languages in CODESYS to understand the differences and even the vulnerabilities introduced during the program compilation process. Based on this analysis, they provide a fuzzing framework (ICSFuzz) to perform security evaluation of the PLC binaries. Our work differs from them because we focus on the source code and do not rely on any specific compilation environment.

Kuzmin et al. [57] propose to use linear-time temporal logic (LTL) to guide program behavior and check whether ST programs satisfy the corresponding temporal logic through Cadence SMV. Darvas et al. [15] propose rule-based reductions and a Cone of Influence (COI) reduction variant for state explosion problems that may be encountered in the formal analysis of ST code, and use the NuSMV model checker to verify temporal logic. After that, they [58] provide a state machine and data-flow-based formal specification method for PLC modules. In addition, they [43] analyze the feasibility of converting between the 5 PLC programming languages provided by Siemens, and point out that the extended SCL (a vendor-defined ST) can be used as the target language for conversion. Adiego et al. [59] propose an intermediate model-based method which can transform PLC programs written in different modeling languages of verification tools to facilitate checking temporal logic. Hailesellasie et al. [60] propose UBIS, which converts ST programs with potential intrusions as well as trusted versions of programs into attributed graphs through UPPAAL, and compares their nodes and edges to detect stealthy code injections. Bohlender et al. [61] apply formal verification and falsification of temporal logic specifications to analyze chemical plant automation systems. Rawlings et al. [62] use symbolic model checking tools st2smv and SynthSMV to verify and falsify a ST program controlling batch reactor systems. Xiong et al. [23] use the behavior model (BM) to specify the behavior of ST programs, and

6. https://bitbucket.org/automforge/matiec_git/issues?status=new&status=open

TABLE 8: The bugs and functional deficiencies of OpenPLC

Type	Problem	Description
Bug	"VAR" parsing exception	The first operation instruction starts with "VAR", and OpenPLC terminates abnormally.
	Division by zero	OpenPLC can check explicit division 0 but allow the execution of implicit division 0.
	Overflow access	OpenPLC can check explicit overflow access but allow the execution of implicit overflow access.
	MOD by zero	OpenPLC provides MOD 0 operation, and the result is 0.
	MOD Exception	The divisor of MOD operation can be empty.
Functional deficiencies	Numerical calculation defects	OpenPLC does not support normal numerical calculation $**$.
		Numerical calculations of `BYTE`, `WORD`, `DWORD` types are not supported.
	Array functions defects	Parentheses are not allowed in array assignments.
	`FUNCTION_BLOCK` instantiation defects	Multiple instantiation of function blocks in one statement is not allowed.
	`ENUM` defects	OpenPLC does not support normal assignment of `ENUM` type.
	Variable declaration defects	Some non-keyword strings cannot be used as variable names, such as "ramp", "LocalVar0_", etc.
		OpenPLC can not support formula and other variables previously declared as initial value.
	Structural defects	Without operation or variable declarations, OpenPLC cannot compile ST program.
		Without statements in `FOR`, `WHILE`, `IF`, `CASE`, `REPEAT`, OpenPLC cannot compile ST.

provide an method based on automatic theoretical to verify LTL attributes on BM. Our work differs from the aforementioned works because they attempt to transform PLC programs into intermediate languages or other programming languages which are suitable for verifying or detecting potential issues, and lack analysis in the conversion process. In addition, these methods do not offer feedback at the level of source code.

Huang et al. [38] is the closest work to ours. They first defined the executable semantics of the ST language in $\mathbb{K}$ and use it to check some security properties. Our work differs because we cover a more complete ST language, and we can use it to discover errors in ST compilers.

7 Conclusion

In this paper, we introduced an executable operational semantics of ST formalized in the $\mathbb{K}$ framework. We presented the semantics of the core features of ST, namely data types, memory operations, its main control statements, and function calls. Our experimental results show that the proposed ST semantics has already covered the main core language features and correctly implements 26,137 lines of public ST code on GitHub. Furthermore, the application of the proposed semantics in testing and analyzing PLC compilers is discussed. By comparing and analyzing the execution results of OpenPLC and K-ST, we found five bugs and some functional deficiencies in OpenPLC. In the future, we hope to further extend K-ST to support the programming environments provided by different vendors. For example, vendors may customize keywords (`Bit STRING` of GX Works2), add additional structures (`LABEL` of Siemens), or even widely extend ST (ExST of CODESYS).

Acknowledgments

We thank the reviewers for their constructive feedback. This research is supported by National Key R&D Program of China under grant 2020YFB2010900, NSFC under grants 61833015 and 62293511, Provincial Key R&D Program of Zhejiang under grants 2020C01038 and 2021C01032, and the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001.

References

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.
[2] G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong, "The 2015 Ukraine blackout: Implications for false data injection attacks," *IEEE Transactions on Power Systems*, vol. 32, no. 4, pp. 3317–3318, 2016.
[3] K. Zetter, "The Ukrainian power grid was hacked again," *Motherboard*, 2017.
[4] N. Perlroth and C. Krauss, "A cyberattack in Saudi Arabia had a deadly goal," *Experts fear another try*, 2018.
[5] D. Tychalas and M. Maniatakos, "Open platform systems under scrutiny: A cybersecurity analysis of the device tree," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2018, pp. 477–480.
[6] A. Nochvay, "Security research: CODESYS runtime, a PLC control framework," *Kaspersky ICS CERT*, 2019.
[7] D. Tychalas, H. Benkraouda, and M. Maniatakos, "ICSFuzz: Manipulating I/Os and repurposing binary code to enable instrumented fuzzing in ICS control applications," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.
[8] "Programmable controllers - Part 3: Programming languages," International Electrotechnical Commission, Standard, 2013.
[9] T. M. Antonsen, *PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming*. BoD–Books on Demand, 2020.
[10] J. O. Blech and S. O. Biha, "On formal reasoning on the semantics of PLC using Coq," *arXiv preprint arXiv:1301.3047*, 2013.
[11] J. O. Blech and S. Ould Biha, "Verification of PLC properties based on formal semantics in Coq," in *International Conference on Software Engineering and Formal Methods*. Springer, 2011, pp. 58–73.
[12] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Software & Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.
[13] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-monitoring for industrial control systems," *Electronics*, vol. 4, no. 4, pp. 995–1017, 2015.
[14] L. Garcia, S. Zonouz, D. Wei, and L. P. De Aguiar, "Detecting PLC control corruption via on-device runtime verification," in *2016 Resilience Week (RWS)*. IEEE, 2016, pp. 67–72.

[15] D. Darvas, B. F. Adiego, A. Vörös, T. Bartha, E. B. Vinuela, and V. M. G. Suárez, "Formal verification of complex properties on PLC programs," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2014, pp. 284–299.

[16] D. Darvas, I. Majzik, and E. B. Viñuela, "Formal verification of safety PLC based control software," in *International Conference on Integrated Formal Methods*. Springer, 2016, pp. 508–522.

[17] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! Attacking PLCs with physical model aware rootkit." in *NDSS*, 2017.

[18] R. Spenneberg, M. Brüggemann, and H. Schwartke, "PLC-Blaster: A worm living solely in the PLC," *Black Hat Asia*, vol. 16, pp. 1–16, 2016.

[19] A. Keliris and M. Maniatakos, "ICSREF: A framework for automated reverse engineering of industrial control systems binaries," *arXiv preprint arXiv:1812.03478*, 2018.

[20] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 326–336.

[21] 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.

[22] G. Canet, S. Couffin, J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in instruction list," in *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics: "Cybernetics Evolving to Systems, Humans, Organizations, and their Complex Interactions"*. IEEE, 2000, pp. 2449–2454.

[23] J. Xiong, X. Bu, Y. Huang, J. Shi, and W. He, "Safety verification of IEC 61131-3 Structured Text programs," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2632–2640, 2020.

[24] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne *et al.*, "Towards automated safety vetting of PLC code in real-world plants," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 522–538.

[25] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg, "Verification of PLC programs given as sequential function charts," in *Integration of software specification techniques for applications in Engineering*. Springer, 2004, pp. 517–540.

[26] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. IEEE, 1999, pp. 106–113.

[27] T. Mertke and G. Frey, "Formal verification of PLC programs generated from signal interpreted Petri nets," in *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat. No. 01CH37236)*, vol. 4. IEEE, 2001, pp. 2700–2705.

[28] R. Huuck, "Semantics and analysis of instruction list programs," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 3–18, 2005.

[29] J. Sadolewski, "Conversion of ST control programs to ANSI C for verification purposes," *e-Informatica Software Engineering Journal*, vol. 5, no. 1, 2011.

[30] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, V. M. G. Suárez, and J. O. Blech, "Modelling and formal verification of timing aspects in large PLC programs," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3333–3339, 2014.

[31] O. Maler and S. Yovine, "Hardware timing verification using KRONOS," in *Proceedings of the Seventh Israeli Conference on Computer Systems and Software Engineering*. IEEE, 1996, pp. 23–29.

[32] M. Heiner and T. Menzel, "Petri net semantics for the PLC user programming language Instruction List," *Techn. Report BTU Cottbus, I-20/1997, Cottbus December*, 1997.

[33] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[34] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[35] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[36] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[37] R. Schumi and J. Sun, "SpecTest: Specification-based compiler testing," *Fundamental Approaches to Software Engineering*, vol. 12649, p. 269, 2021.

[38] Y. Huang, X. Bu, G. Zhu, X. Ye, X. Zhu, and J. Shi, "KST: Executable formal semantics of IEC 61131-3 structured text for verification," *IEEE Access*, vol. 7, pp. 14 593–14 602, 2019.

[39] G. Rosu, "K: A semantic framework for programming languages and formal analysis tools," *Dependable Software Systems Engineering*, vol. 50, p. 186, 2017.

[40] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long, "Evaluation of compiler-induced vulnerabilities," *Journal of Aerospace Information Systems*, vol. 16, no. 10, pp. 409–426, 2019.

[41] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[42] T. R. Alves, M. Buratto, F. M. De Souza, and T. V. Rodrigues, "OpenPLC: An open source alternative to automation," in *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. IEEE, 2014, pp. 585–589.

[43] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Generic representation of PLC programming languages for formal verification," in *23rd PhD Mini-Symposium*. Budapest University of Technology and Economics, 2016, pp. 6–9.

[44] N. Roos, "Programming PLCs using structured text," in *International Multiconference on Computer Science and Information Technology*. Citeseer, 2008, pp. 20–22.

[45] F. Markovic, "Automated test generation for structured text language using uppaal model checker," 2015.

[46] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[47] N. Martı-Oliet and J. Meseguer, "Rewriting logic: roadmap and bibliography," *Theoretical Computer Science*, vol. 285, no. 2, pp. 121–154, 2002.

[48] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 74–91, 2016.

[49] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 533–544, 2012.

[50] D. Bogdanas and G. Roşu, "K-Java: A complete semantics of Java," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 445–456.

[51] D. Park, A. Ştefănescu, and G. Roşu, "KJS: A complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 346–356.

[52] F. Wang, F. Song, M. Zhang, X. Zhu, and J. Zhang, "KRust: A formal executable semantics of Rust," in *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2018, pp. 44–51.

[53] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of Solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[54] T. Nipkow and G. Klein, "Imp: A simple imperative language," in *Concrete Semantics*. Springer, 2014, pp. 75–94.

[55] D. D. McCracken and E. D. Reilly, "Backus-Naur Form (BNF)," in *Encyclopedia of Computer Science*, 2003, pp. 129–131.

[56] G. Roşu and T. F. Şerbănuţă, "K overview and simple case study," *Electronic Notes in Theoretical Computer Science*, vol. 304, pp. 3–56, 2014.

[57] E. V. Kuzmin, A. Shipov, and D. A. Ryabukhin, "Construction and verification of PLC programs by LTL specification," in *2013 Tools & Methods of Program Analysis*. IEEE, 2013, pp. 15–22.

[58] D. Darvas, E. Blanco Vinuela, and I. Majzik, "A formal specification method for PLC-based applications," 2015.

[59] B. F. Adiego, D. Darvas, E. B. Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[60] M. Hailesellasie and S. R. Hasan, "Intrusion detection in PLC-based industrial control systems using formal verification approach in conjunction with graphs," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 1–14, 2018.
[61] D. Bohlender and S. Kowalewski, "Compositional verification of PLC software using horn clauses and mode abstraction," *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 428–433, 2018.
[62] B. C. Rawlings, J. M. Wassick, and B. E. Ydstie, "Application of formal verification and falsification to large-scale chemical plant automation systems," *Computers & Chemical Engineering*, vol. 114, pp. 211–220, 2018.

Kun Wang received the B.S. degree in information and computing sciences from Chongqing University of Posts and Telecommunications of China, in 2017. He received the M.Eng. degree in Cyberspace Security from Xidian University of China, in 2020. He is currently pursuing his Ph.D degree with State Key Laboratory of Industrial Control Technology, Group of Networked Sensing and Control, Zhejiang University. His research interests include control system security and formal methods.

Jingyi Wang is currently a tenure-track assistant professor at the College of Control Science and Engineering, Zhejiang University, China. He received his Ph.D. from Singapore University of Technology and Design in 2018, and his bachelor's degree in Information Engineering from Xi'an Jiaotong University in 2013. He was a research fellow at the School of Computing, National University of Singapore during 2019-2020 and at Information Systems Technology and Design Pillar, Singapore University of Technology and Design during 2018-2019. His research interests include formal methods, software engineering, cyber-security and machine learning.

Christopher M. Poskitt is an Associate Professor of Computer Science (Education) at Singapore Management University (SMU), where he is part of the Centre for Research on Intelligent Software Engineering. Prior to SMU, he held postdoctoral research positions at ETH Zürich and SUTD, and obtained his PhD in Computer Science from the University of York (2014). His research broadly addresses the problem of engineering correct and secure software, especially in the context of cyber-physical systems (e.g. industrial control systems, autonomous vehicles). In addition to software engineering, his research interests span formal methods, cybersecurity, and computer science education.

Xiangxiang Chen received the B.Eng. degree in mechanical engineering from Xi'an Jiaotong University, Xi'an, China in 2021. He is working toward the Ph.D degree in Cyberspace Security at the IS2 Lab at School of Control Science and Engineering, Zhejiang University, Hangzhou, China. His research interests include fuzzing and AI system testing.

Jun Sun is currently a tenured professor at the School of Information Systems, Singapore Management University. He received bachelor's and Ph.D. degrees in computing science from the National University of Singapore (NUS) in 2002 and 2006, respectively. From 2010 to 2019, he was an assistant/associate professor at the Singapore University of Technology and Design. He was a visiting scholar at MIT from 2011 to 2012. His research focuses on software engineering, formal methods, program analysis, and cyber-security. He is the co-founder of the PAT model checker.

Peng Cheng received the B.Sc. degree in automation and the Ph.D. degree in control science and engineering, from Zhejiang University, Hang Zhou, China, in 2004 and 2009, respectively. From 2012 to 2013, he worked as Research Fellow in Information System Technology and Design Pillar, Singapore University of Technology and Design. He is currently a Professor with the College of Control Science and Engineering, Zhejiang University, Hangzhou, China. His research interests include networked sensing and control, cyber-physical systems, and control system security.