Formal Semantics and Analysis of Multitask PLC ST Programs with Preemption

Jaeseo Lee$^{[0000-0001-5979-726X]}$ and Kyungmin Bae$^{(\boxtimes)[0000-0002-6430-5175]}$

Pohang University of Science and Technology, Pohang, South Korea
{sean96, kmbae}@postech.ac.kr

Abstract. Programmable logic controllers (PLCs) are widely used in industrial applications. Ensuring the correctness of PLC programs is important due to their safety-critical nature. Structured text (ST) is an imperative programming language for PLC. Despite recent advances in executable semantics of PLC ST, existing methods neglect complex multitasking and preemption features. This paper presents an executable semantics of PLC ST with preemptive multitasking. Formal analysis of multitasking programs experiences the state explosion problem. To mitigate this problem, this paper also proposes state space reduction techniques for model checking multitask PLC ST programs.

Keywords: PLC · PLC ST · multitask PLC · formal semantics

1 Introduction

Programmable logic controllers (PLCs) are industrial computer systems designed to manage tasks in diverse applications, from assembly lines to robotic devices. The IEC 61131-3 international standard [9] defines the programming languages tailored for developing PLC programs, such as Structured Text (ST), a high-level imperative language. The critical role of PLCs lies in their ability to improve the flexibility, efficiency, and reliability of complex industrial control systems.

Ensuring the correctness of PLC programs is of paramount importance due to their safety-critical nature in industrial applications. Over the years, formal analysis of PLC programs has received significant attention from both academia and industry. In response to this demand, many techniques and tools have been developed for formally analyzing PLC programs, including [10,12,18,2,21,28,5], written in various PLC programming languages.

Recent advances introduce a complete executable semantics of PLC ST [32,17]. Traditional "translation-based" methods (e.g., [12,10,5]) convert PLC ST programs into the input language of another analysis tool. They are inherently limited to a particular syntactic subset of the language, determined by the capabilities of the target input language. In contrast, the complete semantics [32,17] can directly deal with the full syntactic subset of the language.

While the existing complete semantics [32,17,20] detail the language constructs of PLC ST, they overlook complex multitasking aspects. PLC programs run in iterative rounds, called *scan cycles*, interacting with their controlled entities at each iteration. They manage multiple tasks with different periods, deadlines, and priorities, allowing high-priority tasks to preempt low-priority tasks. Capturing this complex nondeterministic behavior remains an unresolved problem.

Our goal is to extend the PLC ST semantics [32,17,20] to cover preemptive multitask programs. There are two central challenges to achieve this goal:

– Although PLC ST programs operate within fixed time intervals, they can be executed or preempted at any moment within a dense time domain. It is essential to completely capture all possible behaviors.
– Task execution and preemption, despite their arbitrariness, often lead to indistinguishable outcomes. It is crucial to identify and focus on the minimal interactions without compromising completeness.

To address these issues, we first define a "time-complete" semantics that naturally captures all possible behaviors over time points. We then introduce an abstraction to identify equivalent behaviors over time intervals.

Our time-complete semantics of PLC ST, based on the K framework [29], explicitly considers each time point within a dense time domain, and faithfully models task execution and preemption at arbitrary times. In particular, a state in our semantics contains a global time, and the `tick` rule can advance the global time by any amount before the deadline caused by intervals of tasks. Since this semantics involves an infinite number of behaviors within a finite time period, it is *not executable* and is unsuitable for automated analysis.

To deal with the non-executability problem, we define an abstraction of the time-complete semantics, resulting in a time-abstract semantics of PLC ST. In contrast to the time-complete semantics, it restricts the focus to a finite number of interleaving scenarios within a finite time period. Each global time in a state is abstracted into the time interval spanning from the earliest start time to the earliest deadline of the tasks involved. Importantly, our semantics is equivalent to the time-complete semantics in terms of bisimulation.

Despite our time-abstract semantics addressing the aforementioned problems, the nondeterministic nature of preemptive multitasking can still lead to the state explosion problem. To illustrate, consider two tasks T_1 and T_2, where T_2 has a higher priority. If T_1 runs a sequence of code $s_1; s_2; \cdots ; s_n$, then T_2 can preempt T_1 after executing any of s_i, resulting in n potential preemption scenarios. However, only statements that interact with global variables can produce different results. To avoid such redundant behavior, we propose state space reduction methods for our semantics, based on partial order reduction [27].

We demonstrate the effectiveness of our time-abstract semantics and state space reduction techniques for automated reachability analysis using several multitask PLC ST benchmarks. The experimental results show a significant improvement in the performance of automated formal analysis using our technique.

This paper is organized as follows. Section 2 gives some background on the basic K semantics of PLC ST and partial order reduction. Section 3 explains details of multitask PLC with preemption and introduces a running example. Section 4 presents the time-complete semantics of PLC ST, and Section 5 presents the time-abstract semantics. Section 6 presents the state space reduction methods for our semantics, and Section 7 shows the experimental results. Section 8 discusses related work. Finally, Section 9 presents some concluding remarks.

2 Preliminaries

K Framework. K [29] is a semantic framework for programming languages, based on rewriting logic [23]. It has been used to formalize large programming languages, including C [11], Java [4], JavaScript [25], and PLC ST [32]. There are several tools that can be used to execute and analyze programming languages using K, including the K tool [19] and Maude [8,31].

In K, program states are represented as multisets of nested cells, called *configurations*. Each cell represents a component of a program state, such as computations and stores. Transitions between configurations are specified as (labeled) K rules, written in a notation that specifies only the relevant parts.

A computation in K is defined as a $\curvearrowright$-separated sequence of computational tasks. For example, $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$ represents the computation consisting of t_1 followed by t_2 followed by t_3, and so on. A task can be decomposed into simpler tasks, and the result of a task is forwarded to the subsequent tasks. E.g., $(5+x)*2$ is decomposed into $x \curvearrowright 5 + \square \curvearrowright \square * 2$, where $\square$ is a placeholder for the result of a previous task. If x evaluates to some value, say 4, then $4 \curvearrowright 5 + \square \curvearrowright \square * 2$ becomes $5 + 4 \curvearrowright \square * 2$, which eventually becomes 18.

The following shows a typical example of K rules for variable lookup, where `lookup` is a label, the k cell contains a computation, *env* contains a map from variables to locations, and *store* contains a map from locations to values:

$$\texttt{lookup:}\ \langle \underset{v}{\underline{x}} \curvearrowright \ldots \rangle_k \ \langle \ldots x \mapsto l \ldots \rangle_{env} \ \langle \ldots l \mapsto v \ldots \rangle_{store}$$

A horizontal line represents a state change, and "..." indicates irrelevant parts. A cell without horizontal lines is not changed by the rule. By the `lookup` rule, if the first item in k is x, then x is replaced by the value v of x in its location l.

PLC ST and its K Semantics. Structured text (ST) is a textual programming language defined in the IEC 61131-3 standard [9]. ST supports common features of imperative programming language, such as (local and global) variable assignments, conditionals, loops, and functions. ST also has unique constructs, such as function blocks, which are callable "objects" with state variables. Functions, function blocks, and programs are called *program organization units* (POUs).

We briefly summarize the syntax of PLC ST. A program is declared with the syntax `PROGRAM` *Name* `...` `END_PROGRAM`. A program consists of variable declarations and code. A variable declaration section is declared with the syntax `VAR` *SectionType* `...` `END_VAR`, where *SectionType* is one of `GLOBAL`, `INPUT`, and `OUTPUT`, or omitted (local in this case). A global variable section can be written outside of a program. A body of code begins after variable sections.

k	**env**	**store**	**stack**	**pouDef**	**pList**
K	$Id \mapsto Loc$	$Loc \mapsto Val$	$List[(Id \mapsto Loc) \times K]$	$Pid \mapsto VarDecl \times Code$	$List[Pid]$

Fig. 1. Examples of K cells for PLC ST.

`assign`: $\langle \frac{x := v}{\cdot} \curvearrowright \ldots\rangle_k \ \langle \ldots x \mapsto l \ldots\rangle_{env} \ \langle \ldots l \mapsto \frac{\_}{v} \ldots\rangle_{store}$

`if-true`: $\frac{\texttt{IF}\ \mathit{true}\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2\ \texttt{END\_IF}}{s_1}$ `if-false`: $\frac{\texttt{IF}\ \mathit{false}\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2\ \texttt{END\_IF}}{s_2}$

`fbCall`: $\langle \frac{f(args)}{bd(args, D) \curvearrowright ex(\eta, S)} \curvearrowright \ldots\rangle_k \ \langle \ldots f \mapsto l \ldots\rangle_{env} \ \langle \ldots l \mapsto [fb, \eta] \ldots\rangle_{store} \ \langle \ldots fb \mapsto (D, S) \ldots\rangle_{pouDef}$

`fbExec`: $\langle \frac{ex(\eta, S) \curvearrowright \kappa}{S}\rangle_k \ \langle \frac{\cdot}{(\rho, \kappa)} \ldots\rangle_{stack} \ \langle \frac{\rho}{\eta}\rangle_{\mathrm{env}}$ `fbQuit`: $\langle \frac{\cdot}{\kappa}\rangle_k \ \langle \frac{\_}{\rho}\rangle_{env} \ \langle \frac{(\rho, \kappa)}{\cdot} \ldots\rangle_{stack}$

Fig. 2. Examples of K rules for PLC ST.

We give an overview of the K semantics of PLC ST [17,20,32]. Figure 1 shows part of the structure of K configurations. The *k*, *env*, and *store* cells are explained above. The *stack* cell contains a call stack, which stores the caller's environment and computation when a function block is called. The *pouDef* cell is a map from POU identifiers to POU declarations, each of which contains variable declarations and code. The *pList* cell contains a list of programs to run.

Figure 2 shows some of the K rules in the PLC ST semantics. Thanks to the modularity of the K technique [29,30], the K rules for common imperative language constructs, such as `assign` for variable assignment, and `if-true` and `if-false` for conditional statements, are (almost) identical to those for other imperative languages, except for slight syntactic differences.

When a function block is called (`fbCall`), the POU instance $[fb, \eta]$ is obtained, where η is a local environment, and tasks of binding the arguments and executing the code S are loaded in k. When the code S is executed (`fbExec`), the current environment ρ and the remaining computation κ are pushed to *stack*, and η becomes a new environment in *env*. When there are no more tasks in k (`fbQuit`), the previous environment ρ and the computation κ are restored from *stack*.

Transition Systems. A *transition system* $\mathcal{S}$ is a tuple (S, s_0, T, AP, L) [1,27], where S is a set of states, $s_0 \in S$ is an initial state, T is a set of transitions such that $\alpha \in T$ is a partial function $\alpha : S \to S$, AP is a set of atomic propositions, and $L : S \to 2^{AP}$ is a state labeling function. A transition $\alpha \in T$ is *enabled* in a state $s \in S$ if $\alpha(s)$ is defined. We denote by *enabled*(s) the set of transitions enabled in s. We often write $s \xrightarrow{\alpha} s'$ to denote $\alpha(s) = s'$ for $s, s' \in S$.

For two transition systems $\mathcal{S}_i = (S_i, s_0^i, T_i, AP, L_i)$, $i = 1, 2$, a binary relation $R \in S_1 \times S_2$ is a *simulation* [7] from $\mathcal{S}_1$ to $\mathcal{S}_2$ iff: (i) $(s_0^1, s_0^2) \in R$; and (ii) for any $(s_1, s_2) \in R$, $L(s_1) = L(s_2)$ holds, and if $s_1 \xrightarrow{\alpha} s_1'$, there exists $s_2' \in S$ such that $s_2 \xrightarrow{\alpha} s_2'$ and $(s_1', s_2') \in R$. A simulation R from $\mathcal{S}_1$ to $\mathcal{S}_2$ is called a *bisimulation* iff R^{-1} is also a simulation from $\mathcal{S}_2$ to $\mathcal{S}_1$.

The K semantics of PLC ST naturally defines a transition system, provided that AP and L are given. States are given by K configurations. Each transition α_l is identified by a rule label l such that $s \xrightarrow{\alpha_l} s'$ iff s is reduced to s' by a K rule with label l. For the "single-task" case, α_l is well defined as a partial function because single-task PLC programs are deterministic. For the "multitask" case, we also need task identifiers as well as rule labels (see Section 4.2).

Partial Order Reduction. Consider a transition system $\mathcal{S} = (S, s_0, T, AP, L)$. A transition $\alpha \in T$ is *invisible* iff $s \xrightarrow{\alpha} s'$ implies $L(s) = L(s')$. An *independence relation* $I \subseteq T \times T$ is a symmetric and anti-reflexive relation such that for any pair of transitions $(\alpha, \beta) \in I$ and state $s \in S$, where $\alpha, \beta \in \mathit{enabled}(s)$, (i) $\alpha \in \mathit{enabled}(\beta(s))$ and $\beta \in \mathit{enabled}(\alpha(s))$, and (ii) $\alpha(\beta(s)) = \beta(\alpha(s))$. Its complement $D = (T \times T) \setminus I$ is called a dependency relation.

We consider partial order reduction using ample sets [27]. An *ample set* of a state $s \in S$ is a subset of the enabled transitions $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$. A state $s \in S$ is called *fully expanded* when $\mathit{ample}(s) = \mathit{enabled}(s)$. When exploring the state space, only the transitions in $\mathit{ample}(s)$ are explored instead of all the transitions in $\mathit{enabled}(s)$. This results in a reduced transition system $\hat{\mathcal{S}}$ that is behaviorally equivalent when ample sets are chosen appropriately.

The following conditions guarantee that a transition system $\mathcal{S}$ and its reduced version $\hat{\mathcal{S}}$ are behaviorally equivalent [27]: (i) $\mathit{ample}(s) \neq \emptyset$ iff $\mathit{enabled}(s) \neq \emptyset$; (ii) a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot occur before a transition in $\mathit{ample}(s)$ occurs first.[1]; (iii) if s is not fully expanded, all transitions in $\mathit{ample}(s)$ are invisible; and (iv) any cycle in the reduced state space $\hat{\mathcal{S}}$ contains at least one fully expanded state.

3 Multitask PLC and a Running Example

In multitask PLC, each program is assigned an *interval* and a *priority*. A program is scheduled to run periodically, where the interval determines the duration of each period. Priorities are given as natural numbers, where a lower number indicates a higher priority. A program with a higher priority can preempt the execution of a program with a lower priority. The execution of each program must be completed before the beginning of its next round.

Multitask PLC programs are difficult to analyze because of their complex interleaving possibilities. The number of different executions caused by different interleaving increases exponentially with the number of programs due to the nondeterministic nature of preemption. This is because context switches can occur at any point of each program. E.g., consider a program with k statements. If the program is preempted by another program, the preemption can occur after the execution of the i-th statement for any $1 \leq i \leq k$. Therefore, for n programs with different priorities, there are $O(k^{n-1})$ interleaving possibilities by preemption.

[1] For $s \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$, if α depends on $\mathit{ample}(s)$, $\beta_i \in \mathit{ample}(s)$ for some $i \leq n$.

Our running example is inspired by the two-wheeled self-balancing robot.[2] The robot moves on flat ground while maintaining its balance. It is equipped with a sonar sensor that detects nearby obstacles. The current state information is displayed on the attached panel. It takes control input from a remote controller to move forward, backward, and turn.

The system consists of three programs: `balanceControl`, `sonar`, and `display`, with intervals of 3 ms, 4 ms, and 12 ms, and priorities of 1, 2, and 3, respectively. Figure 3 shows a code snippet of `balanceControl` and `sonar` (manually translated from the original source), where intervals and priorities are declared in the `CONFIGURATION` block. There are two global variables `mode` and `obstacle_flag`, which are used for communication between different programs.

The `balanceControl` program takes control inputs (such as `cmd_forward` and `cmd_turn`) and balancing inputs (such as `gyro_sensor`). The robot has two modes CAL and CONTROL, where the global variable `mode` indicates the current mode. When `balanceControl` is executed for the first time, it calibrates and sets the appropriate initial settings for the robot and sets `mode` to `CONTROL`. The program starts controlling the robot from the second round.

The `sonar` program takes sonar sensor inputs (such as `sonar`). When `mode` is CONTROL, the program measures the distances to nearby objects to detect an imminent collision hazard. If so, it sets the global variable `obstacle_flag` to `TRUE`. At this point, `balanceControl` ignores its control input and attempts to stop the robot by setting `cmd_forward` to -100.

Figure 4 shows two interleaving scenarios that reach different outcomes. Each rectangle denotes the range from the *earliest possible start time* to the *deadline* for a task. The heads and tails of horizontal arrows denote the start and end of program execution. The curved vertical arrows denote preemption and its return. In Scenario 1, there is no preemption and `mode` is definitely `CAL` during the first execution of `sonar`. In Scenario 2, the execution of `sonar` is preempted by `balanceControl` and `mode` may become `CONTROL`, in which case `sonar` sets `obstacle_flag` to 1.

4 Formal Semantics of Multitask PLC

This section presents an executable semantics of PLC ST with preemptive multitasking, which extends the existing K semantics of PLC ST [17,20,32] explained in Section 2. Our semantics specifies all possible interleaving scenarios by nondeterministic preemption over a dense time domain. We explicitly take into account a global time that can be advanced by any amount up to the deadline, which is determined by the intervals of tasks.

4.1 K Configuration for Multitask PLC

Figure 5 depicts the K cells for specifying preemptive multitasking behaviors, in addition to the existing cells in the original semantics [17,20,32]: (1) *time*

[2] `https://lejos-osek.sourceforge.net/nxtway_gs.htm`

```
CONFIGURATION Config
  RESOURCE Res ON PLC
    TASK T1(INTERVAL:= T#3ms, PRIORITY := 1);
    TASK T2(INTERVAL:= T#4ms, PRIORITY := 2);
    TASK T3(INTERVAL:= T#12ms, PRIORITY := 3);
    PROGRAM bCtrl WITH T1 : balanceControl();
    PROGRAM sonar WITH T2 : sonar();
    PROGRAM dsply WITH T3 : display();
  END_RESOURCE
  VAR_GLOBAL
    mode : MODE := CAL ;
    obstacle_flag : BOOL := FALSE ;
  END_VAR
END_CONFIGURATION

PROGRAM balanceControl
  VAR_INOUT
    cmd_forward : DINT ;
    cmd_turn : DINT ;
    gyro_sensor : REAL ;
    ...
  END_VAR

  VAR
    avg_cnt : REAL := 0 ;
    ...
  END_VAR

  CASE mode OF
```

```
  CAL:
    gyro_offset := gyro_offset + gyro_sensor;
    avg_cnt := avg_cnt + 1;
    IF avg_cnt >= 1 THEN
     gyro_offset := gyro_offset / avg_cnt;
     mode := CONTROL;
     ...
    END_IF;
  CONTROL:
    IF obstacle_flag THEN
      cmd_forward := -100;
      ...
    END_IF;
  END_CASE;
END_PROGRAM

PROGRAM sonar
  VAR_INPUT
    sonar : REAL;
    ...
  END_VAR

  IF mode = CONTROL AND sonar <= 100 THEN
    obstacle_flag := 1;
    ...
  END_IF;
  ...
END_PROGRAM
...
```

Fig. 3. Two-wheeled self-balancing robot code.

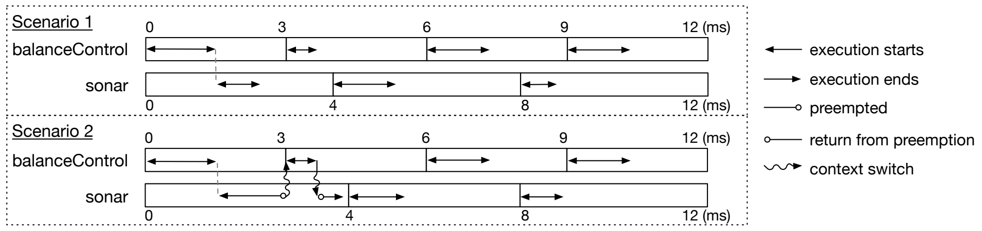

Fig. 4. Two interleaving scenarios of the robot example.

denotes the current time; (2) *active* denotes the identifier of the currently running program; (3) *interval* has a map from program identifiers to their intervals; (4) *pQueue* contains a priority queue of tasks that are ready to run according to *time* and *interval*; and (5) *futureTS* contains tasks that are not ready.

Tasks are represented as a tuple (id, pr, es, dl), where id is the identifier of the program, pr is the program's priority, es is the earliest start time, and dl is the deadline. Each program can start after its earliest start time and must end before its deadline. When the current time is 0, es is 0 and dl is the interval.

The *Program* cell encompasses the program's identifier, a computation, and an environment and a call stack. Unlike the single-task semantics in Section 2, in our multi-task semantics, each program maintains its own computation, environment, and stack. That is, a full K configuration has the nested structure of the form

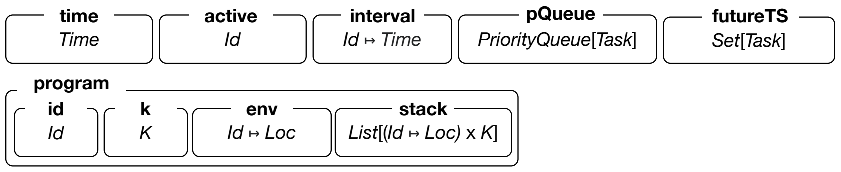

Fig. 5. K configurations for multitask PLC.

tick: $\langle \frac{t}{t'} \rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft\rangle_{futureTS}$ where $t \leq t' < minDL(pq, ft)$

execute: $\langle \frac{\cdot}{P} \rangle_{active}\ \langle (P, \_, es, dl) \ldots\rangle_{pQueue}\ \langle\langle P\rangle_{id}\ \langle \frac{\blacksquare}{\cdot} \curvearrowright \ldots\rangle_{k}\ \ldots\rangle_{program}$

placeT: $\langle t\rangle_{time}\ \langle \frac{pq}{insert(pq, (P, pr, t, dl))} \rangle_{pQueue}\ \langle \ldots \frac{(P, pr, es, dl)}{\cdot} \ldots\rangle_{futureTS}$ where $\{t\} \cap [es, dl) \neq \emptyset$

endProgram: $\langle t\rangle_{time}\ \langle \ldots P \mapsto \iota \ldots\rangle_{interval}\ \langle\langle P\rangle_{id}\ \langle \frac{\cdot}{\blacksquare \curvearrowright code} \rangle_{k}\rangle_{program}\ \langle \ldots \frac{(P, pr, es, dl)}{\cdot} \ldots\rangle_{pQueue}$

$\langle \ldots \frac{\cdot}{(P, pr, es + \iota, dl + \iota)} \ldots\rangle_{futureTS}\ \langle \frac{P}{\cdot} \rangle_{active}\ \langle \ldots P \mapsto (\_, code) \ldots\rangle_{pouDef}$ where $es < t$

preempt: $\langle \frac{P'}{P} \rangle_{active}\ \langle (P, pr, es, dl) \ldots (P', pr', es', dl') \ldots\rangle_{pQueue}$

$\langle\langle P\rangle_{id}\ \langle \frac{\blacksquare}{\cdot} \curvearrowright \ldots\rangle_{k}\rangle_{program}\ \langle\langle P'\rangle_{id}\ \langle \frac{\cdot}{\blacksquare} \curvearrowright \ldots\rangle_{k}\rangle_{program}$

Fig. 6. K rules for preemptive multitasking.

(where other K cells not used in this paper are omitted):

$$\langle \ldots\rangle_{time}\ \langle \ldots\rangle_{active}\ \langle \ldots\rangle_{interval}\ \langle \ldots\rangle_{pQueue}\ \langle \ldots\rangle_{futureTS}\ \langle \ldots\rangle_{store}\ \langle \ldots\rangle_{pouDef}\ \cdots$$
$$\langle\langle \ldots\rangle_{id}\ \langle \ldots\rangle_{k}\ \langle \ldots\rangle_{env}\ \langle \ldots\rangle_{stack}\ \ldots\rangle_{program}\ \cdots\ \langle\langle \ldots\rangle_{id}\ \langle \ldots\rangle_{k}\ \langle \ldots\rangle_{env}\ \langle \ldots\rangle_{stack}\ \ldots\rangle_{program}$$

4.2 K Rules for Multitask PLC

Figure 6 shows the K rules to specify preemptive multitasking behaviors. The `tick` rule (nondeterministically) increments the current time (in the *time* cell) to an arbitrary time before the minimum deadline of the tasks. The *minDL* function is defined as follows, where $deadlines(A)$ denotes the set of deadlines in A:

$$minDL(pq, ft) = \min(deadlines(pq) \cup deadlines(ft))$$

The `execute` rule executes the top task in *pQueue* if no task is currently running. Before the rule is applied, the *active* cell is empty, and the execution of each program is "blocked" by ■ at the top of its *k* cell. Suppose P is the program for the top task in *pQueue*. When `execute` is applied, the *active* cell is updated with the program's identifier P, and ■ is removed from the top of P's *k* cell.

The `placeT` rule moves a task in *futureTS* into *pQueue*, when the task is ready to run according to *time* and *interval*. The function $insert(pq, T)$ inserts task T into the priority queue *pq*. The side condition states that the current time is between its earliest start time and the deadline. It also sets the third item of the task to the current time t to record when this happens.

The `endProgram` rule is applied when the execution of the active program is finished. Suppose P is the active program and the k cell of P is empty. When `endProgram` is applied, the *active* cell becomes empty and the corresponding task is removed from *pQueue*. The subsequent task for P is added to *futureTS*, where the earliest start time and deadline are increased by P's interval ι. Finally, the code of P, where the execution is blocked by ■, is loaded into the k cell of P. The side condition asserts that the execution of P takes non-zero time.

The `preempt` rule preempts a lower-priority task, and executes a higher-priority task in *pQueue*. In the rule, P has a higher priority than P' because it is the top element in *pQueue*. The *active* becomes P, and ■ moves to P' from P.

The side condition of `tick` maintains the validity constriaint: the value of the *time* cell should not exceed any of the deadlines of the tasks in *pQueue* and *futureTS*, as shown in the following lemma.

Lemma 1. *For a K configuration that satisfies the validity constraint, any next configuration obtained by applying a rule also satisfies the constraint.*

Proof. All the rules except `tick`, including `lookup`, `assign`, `if-T`, `if-F`, `fbCall`, `execute`, `placeT`, `preempt`, and `endProgram` do not modify the *time* cell. For such a rule r, if $s \xrightarrow{r} s'$ then the time value of s is equal to the time value of s'.

For `tick`, its side condition keeps the new time t to not exceed any of the deadlines of tasks in *pQueue* and *futureTS*. □

It is worth noting that the rules in Figure 6 are all nondeterministic. The time can be increased by any value up to the deadline, and different tasks can have the same priority and interval. For this reason, transitions with `tick` are identified by time differences (e.g., `tick`(1) increases the time by 1), and transitions with the other four rules are identified by rule labels and program identifiers (e.g., `placeT(`P`)` moves a task $(P, \ldots)$ from *futureTS* to *pQueue*).

4.3 Example of K Rule Applications

Figure 7 shows a sequence of states simulating an execution path for Scenario 2 in Figure 4. Applying `execute` to state s_1 to execute `balanceControl` gives s_2. After 1 second, `balanceControl` executes its code (using other K rules) and then `endProgram` is applied, resulting in s_3. The following shows the transitions:

$$s_1 \xrightarrow{\texttt{execute}(P_b)} s_2 \xrightarrow{\texttt{tick}(1)} \cdots \xrightarrow{\texttt{endProgram}(P_b)} s_3$$

$$\xrightarrow{\texttt{execute}(P_s)} \cdots \xrightarrow{\texttt{tick}(2)} \cdots \xrightarrow{\texttt{placeT}(P_b)} s_4 \xrightarrow{\texttt{preempt}(P_b)} s_5 \longrightarrow \cdots$$

s_1	$\langle 0\rangle_{time}\ \langle\cdot\rangle_{active}\ \langle(P_b,1,0,3),(P_s,2,0,4)\rangle_{pQueue}$ $\langle\cdot\rangle_{futureTS}\ \langle\langle P_b\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program}\ \langle\langle P_s\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program} \cdots$
s_2	$\langle 0\rangle_{time}\ \langle P_b\rangle_{active}\ \langle(P_b,1,0,3),(P_s,2,0,4)\rangle_{pQueue}$ $\langle\cdot\rangle_{futureTS}\ \langle\langle P_b\rangle_{id}\ \langle\ldots\rangle_k \cdots\rangle_{program}\ \langle\langle P_s\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program} \cdots$
s_3	$\langle 1\rangle_{time}\ \langle\cdot\rangle_{active}\ \langle(P_s,2,0,4)\rangle_{pQueue}$ $\langle(P_b,1,3,6)\rangle_{futureTS}\ \langle\langle P_b\rangle_{id}\ \langle\cdot\rangle_k \cdots\rangle_{program}\ \langle\langle P_s\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program} \cdots$
s_4	$\langle 3\rangle_{time}\ \langle P_s\rangle_{active}\ \langle(P_b,1,3,6),(P_s,2,0,4)\rangle_{pQueue}$ $\langle\cdot\rangle_{futureTS}\ \langle\langle P_b\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program}\ \langle\langle P_s\rangle_{id}\ \langle\ldots\rangle_k \cdots\rangle_{program} \cdots$
s_5	$\langle 3\rangle_{time}\ \langle P_b\rangle_{active}\ \langle(P_b,1,3,6),(P_s,2,0,4)\rangle_{pQueue}$ $\langle\cdot\rangle_{futureTS}\ \langle\langle P_b\rangle_{id}\ \ldots\rangle_k \cdots\rangle_{program}\ \langle\langle P_s\rangle_{id}\ \langle\blacksquare \curvearrowright \ldots\rangle_k \cdots\rangle_{program} \cdots$

Fig. 7. An example of execution sequences, where $P_b = \mathit{balanceControl}$ and $P_s = \mathit{sonar}$.

Likewise, Scenario 1 can be simulated by the following sequence of transitions. It is the same as the above up to s_3, and has different states after that.

$$s_1 \xrightarrow{\texttt{execute}(P_b)} s_2 \xrightarrow{\texttt{tick}(1)} \cdots \xrightarrow{\texttt{endProgram}(P_b)} s_3$$
$$\xrightarrow{\texttt{execute}(P_s)} \cdots \xrightarrow{\texttt{tick}(2)} \cdots \xrightarrow{\texttt{endProgram}(P_s)} s_4' \xrightarrow{\texttt{placeT}(P_b)} s_5' \xrightarrow{\texttt{execute}(P_b)} \cdots$$

5 Time Abstraction

A single program execution can produce an infinite number of cases, due to *nondeterministic time advances*. For example, in Scenario 2 of Figure 4, the first execution of `balanceControl` can end in 1 *ms*, 0.5 *ms*, 0.25 *ms*, 0.125 *ms*, and so on. However, there are only a finite number of critical times that may change the possible behaviors.

This section presents a time abstraction for our multitask PLC ST semantics. The main idea is to express time abstractly with a time interval that represents an infinite number of time points. We define an abstract function that maps a concrete K configuration to its abstract version and apply it globally to the K rules defined in Section 4.2. We show that the resulting abstract semantics is equivalent to the concrete PLC semantics in terms of bisimulation.

5.1 Abstraction function

The abstraction function takes a K configuration with a time value and returns the K configuration with a time interval that (i) contains the original time, and (ii) encompasses all other times that have equivalent behaviors. Let *maxES* be defined as follows, where *startTimes*(*pq*) is the set of earliest start times in *pq*:

$$\mathit{maxES}(pq) = \min(\mathit{startTimes}(pq))$$

Definition 1. *Given a K configuration* $s = \langle t\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft\rangle_{futureTS}\ \cdots$ *that satisfies the validity constraint, its* time abstraction *is defined as follows, where* $abs(t, pq, ft) = \langle max(maxES(pq), t), minDL(pq,\ ft)\rangle$*:*

$$\lambda(s) = \langle\!|\ abs(t, pq, ft)\ |\!\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft\rangle_{futureTS}\ \cdots$$

`placeT'`: $\langle| \frac{t_{min}}{max(t_{min}, es)}, t_{max} |\rangle_{time}\ \langle \frac{pq}{insert(pq, (P, pr, es, dl))} \rangle_{pQueue}\ \langle \dots \frac{(P, pr, es, dl)}{\cdot} \dots \rangle_{futureTS}$

where $[t_{min}, t_{max}) \cap [es, dl) \neq \emptyset$

`endProgram'`: $\langle| \frac{t_{min}}{max(maxES(pq'), es)}, \frac{t_{max}}{minDL(pq', ft')} |\rangle_{time}\ \langle \dots P \mapsto \iota \dots \rangle_{interval}\ \langle \frac{P}{\cdot} \rangle_{active}\ \langle \frac{pq}{pq'} \rangle_{pQueue}$

$\langle \frac{ft}{ft'} \rangle_{futureTS}\ \langle \langle P \rangle_{id}\ \langle \frac{\cdot}{\blacksquare \curvearrowright code} \rangle_{k} \rangle_{program}\ \langle \dots P \mapsto (\_, code) \dots \rangle_{pouDef}$

where $(P, pr, es, dl) \in pq$, and $pq' = pq \setminus \{(P, pr, es, dl)\}$, and $ft' = ft \cup \{(P, pr, es + \iota, dl + \iota)\}$

Fig. 8. K rules for multitask interleaving with abstract time

Now the *time* cell contains a pair of times $|\ t_1, t_2\ |$, and represents the set of all the times that are contained in the left-closed right-open interval $[t_1, t_2)$.

Figure 8 shows the interleaving rules with the abstract time. Except `tick`, `endProgram`, and `placeT`, all other K rules, including `execute` and `preempt`, are the same before and after the abstraction. The `tick` rule is now identity. The `endProgram` and `placeT` rules move the possible time range of the system. It moves the minimum time value (left) to *maximum earliest start times* of the tasks in *pQueue* or remains unchanged if the priority queue is empty. The maximum time value (right) is moved to the *minimum deadlines* of the tasks in *pQueue* and *futureTS* altogether.

5.2 Equivalence Before and After Abstraction

The concrete semantics and the abstract semantics are equivalent in terms of bisimulation. Let R be a binary relation between concrete configurations and abstract configurations such that $(s, \lambda(s)) \in R$ for each configuration s. Then, R is a bisimulation with respect to atomic propositions not depending on *time*.

By construction, for a concrete transition $s \xrightarrow{\alpha} s'$, there exists an abstract transition $\lambda(s) \xrightarrow{\alpha} \lambda(s')$. For an abstract transition $\hat{s} \xrightarrow{\alpha} \hat{s'}$, there also exists a corresponding concrete transition $s \xrightarrow{\alpha} s'$, where the *time* values t and t' of s and s', respectively, can be any values in the corresponding intervals such that: (i) $t \leq t'$ if $\alpha = \texttt{tick}$, and (ii) $t = t'$ if $\alpha \neq \texttt{tick}$.

Theorem 1. *Given an initial K configuration s_0 satisfying the validity constraint, R is a bisimulation between the concrete transition system $\mathcal{S}$ from s_0 and the abstract transition system $\hat{\mathcal{S}}$ from $\lambda(s_0)$.*

Proof. To show bisimilarity, we need to prove the following conditions mentioned in Section 2: (i) $(s_0, \lambda(s_0)) \in R$ (ii) for any s, $L(s) = L(\lambda(s))$ and $s \xrightarrow{\alpha} s'$ iff $\lambda(s) \xrightarrow{\alpha} \lambda(s')$.

Condition (i) automatically holds since the initial state of abstract semantics $(s, \lambda(s)) \in R$ by the definition of R. The former part of condition (ii), which is $L(s) = L(\lambda(s))$ for any s holds since we assume that the labeling function L has nothing to do with the *time* cell. For the latter part of condition (ii), let us first consider the transitions that do not use the *time* cell such as `lookup`, `fbCall`, `if-true`, `if-false`, `fbExec`, and `fbQuit`. They are identical in both transition systems. Thus, if $s \xrightarrow{r} t$ holds for any such rule r, then, $\lambda(s) \xrightarrow{r} \lambda(t)$ and vice versa. The transitions that use *time* cells are $\texttt{tick}(\tau)$, $\texttt{placeT}(P)$, and $\texttt{endProgram}(P)$. For each of these transitions, we show if $s \xrightarrow{r} t$ holds for any such rule r, then, $\lambda(s) \xrightarrow{r} \lambda(t)$ and vice versa.

When $\alpha = \texttt{tick}(\tau)$, assume $a \xrightarrow{\texttt{tick}(\tau)} b$, where $a = \langle t\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ts\rangle_{futureTS} \ldots$ and $b = \langle t'\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ts\rangle_{futureTS} \ldots$. If $\lambda(a) \xrightarrow{\texttt{tick'}} b'$, then since `tick'` is identity, $b' = \lambda(a) = \langle\!|\ max(maxES(pq), t), minDL(pq, ts)\ |\!\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ts\rangle_{futureTS} \ldots$, which is equal to $\lambda(b)$. Conversely, assume $a' \xrightarrow{\texttt{tick'}} b'$, where $a' = \langle\!|\ t_1, t_2\ |\!\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ts\rangle_{futureTS}\ \ldots = b'$. Take a and b to be $\langle t_1\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ts\rangle_{futureTS}\ \ldots$ and $a \xrightarrow{\texttt{tick}(0)} b$ holds.

When $\alpha = \texttt{placeT}(P)$, assume $a \xrightarrow{\texttt{placeT}(P)} b$, where $a = \langle t\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft_1\ (P, pr, es, dl)\ ft_2\rangle_{futureTS} \ldots$ and $b = \langle t\rangle_{time}\ \langle insert(pq, (P, pr, es, dl))\rangle_{pQueue}\ \langle ft_1\ ft_2\rangle_{futureTS} \ldots$. $\lambda(a) = \langle\!|\ max(maxES(pq), t), minDL(pq, ts)\ |\!\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft_1\ (P, pr, es, dl)\ ft_2\rangle_{futureTS} \ldots$. To satisfy the rule condition of `placeT'`, we must show that $[max(maxES(pq), t), minDL(pq, ts)) \cap [es, dl) \neq \emptyset$.

By the rule condition of `placeT`, any earliest start times put in *pQueue* are less than or equal to t. Thus, $max(maxES(pq), t) \leq t$ holds. In addition, $es \leq t$ holds since $a \xrightarrow{\texttt{placeT}(P)} b$. By Lemma 1, $t < minDL(pq, ft_1\ (P, pr, es, dl)\ ft_2)$ holds. It is also trivial that $minDL(pq, ft_1\ (P, pr, es, dl)\ ft_2) \leq dl$. Putting these inequalities together, $max(maxES(pq), t) \leq t < minDL(pq, ft_1\ (P, pr, es, dl)\ ft_2) \leq dl$, and $es \leq t < minDL((pq, ft_1\ (P, pr, es, dl))$. This satisfies the rule condition of `placeT'` and $\lambda(a) \xrightarrow{\texttt{placeT'}(P)} \langle\!|\ max(maxES(pq), t), minDL(pq, ts)\ |\!\rangle_{time}\ \langle insert(pq, (P, pr, es, dl))\rangle_{pQueue}\ \langle ft_1\ ft_2\rangle_{futureTS} \ldots = \lambda(b)$.

Conversely, assume $a' \xrightarrow{\texttt{placeT'}(P)} b'$, where $a' = \langle\!|\ t_1, t_2\ |\!\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft_1\ (P, pr, es, dl)\ ft_2\rangle_{futureTS}\ \ldots$ and $b' = \langle\!|\ t_1, t_2\ |\!\rangle_{time}\ \langle insert(pq, (P, pr, t, dl))\rangle_{pQueue}\ \langle ft_1\ ft_2\rangle_{futureTS}\ \ldots$. By the rule condition of $\texttt{placeT'}(P)$, $[t_1, t_2) \cap [es, dl) \neq \emptyset$. We pick any number $t \in [t_1, t_2) \cap [es, dl)$. Then, let $a = \langle t\rangle_{time}\ \langle pq\rangle_{pQueue}\ \langle ft_1\ (P, pr, es, dl)\ ft_2\rangle_{futureTS}\ \ldots$ and assume $\lambda(a) = a'$. Then, since $t \in [es, dl)$, $a \xrightarrow{\texttt{placeT}(P)} b$, where $b = \langle t\rangle_{time}\ \langle insert(pq, (P, pr, t, dl))\rangle_{pQueue}\ \langle ft_1\ ft_2\rangle_{futureTS}\ \ldots$ and $\lambda(b) = b'$.

When $\alpha = \texttt{endProgram}(P)$, Assume $a \xrightarrow{\texttt{endProgram}(P)} b$, then trivially $\lambda(a) \xrightarrow{\texttt{endProgram'}(P)} \lambda(b)$ holds, because there is no side condition on `endProgram'` and the top-side pattern is the same except *time* cell. Conversely, assume $a' \xrightarrow{\texttt{endProgram'}(P)} b'$, where $a' = \langle\!|\ t_1, t_2\ |\!\rangle_{time}\ \langle P\rangle_{active}\ \langle \ldots\ (P, pr, es, dl)\ \ldots\rangle_{pQueue}\ \ldots$, and $b' = \langle\!|\ t_1, t_2\ |\!\rangle_{time}\ \langle\cdot\rangle_{active}\ \langle\ldots\rangle_{pQueue}\ \ldots$ We already showed that $es \leq t_1 < t_2$ holds when $\alpha = \texttt{placeT}(P)$. Let a be $\langle(t_1+t_2)/2\rangle_{time}\ \langle P\rangle_{active}\ \langle\ldots\ (P, pr, es, dl)\ \ldots\rangle_{pQueue}\ \ldots$ and assume $\lambda(a) = a'$. Trivially, $es < (t_1 + t_2)/2$ holds. Thus, $a \xrightarrow{\texttt{endProgram}(P)} b$, where $b = \langle(t_1 + t_2)/2\rangle_{time}\ \langle\cdot\rangle_{active}\ \langle\ldots\rangle_{pQueue}\ \ldots$ holds and $\lambda(b) = b'$. □

6 State Space Reduction

In this section, we introduce two state space reduction methods that reduce the state space. Since the K rules we introduced involve many nondeterministic choices, it results in a large state space that makes it hard to analyze.

The first technique is the application of the ample set approach. Based on the observation that interleaving of `placeT` with other rules spawns many different but essentially the same execution paths, we include `placeT` in the ample set, if not fully expanded. It is not simple because not all enabled `placeT` can be prioritized without consequences. The second technique is to put rules that do not change the local memory first. This is possible because the atomic properties of interest in this paper only depend on the local memory.

6.1 Our Ample Set Approach

Consider state $\langle\cdot\rangle_{active}$ $\langle(P_1, 1, 0, 20)\rangle_{pQueue}$ $\langle(P_2, 2, 5, 25)\rangle_{futureTS}$ .... From this state, both `execute` and `placeT` are applicable. In either order, it converges to $\langle P\rangle_{active}$ $\langle(P_1, 1, 0, 20)\ \langle(P_2, 2, 5, 25)\rangle_{pQueue}$ $\langle\cdot\rangle_{futureTS}$ ..., and all states in this procedure including the intermediate states share the same set of atomic properties held. Thus, we only need to explore one of these paths. This phenomenon stems from the independence of these two rules. Just like this case, when `placeT` does not change the top element of *pQueue*, it is only dependent on `tick`.

Definition 2. *For a state s, if* $\mathsf{placeT}(P) \in$ *enabled*(s) *and* $\mathsf{placeT}(P)$ *does not change the top element of pQueue, ample*$(s) = \{\mathsf{placeT}(P), \mathsf{tick}(\tau)\}$*; otherwise, ample*$(s) =$ *enabled*(s)*.*

To prove that Definition 2 satisfies the ample set conditions, we first show two lemmas to help with proving that condition (ii) holds. Lemma 2 proves the independence of `placeT` and others when it does not change the top element of *pQueue*. This is because only the top element of *pQueue* decides what to execute or preempt. Lemma 2 also shows the independence of `tick` and other rules except `placeT`, since other rule applications do not restrain the side condition of `tick`.

Lemma 2. *(1)* $\mathsf{placeT}(P)$ *is independent with all other transitions except* $\mathsf{tick}(\tau)$*, if it does not change the top element of the pQueue cell. (2)* $\mathsf{tick}(\tau)$ *is independent from all other transitions except* $\mathsf{placeT}(\cdot)$*.*

Proof. (1) Except for $\mathsf{tick}(\tau)$, we need to show the independence of all transitions with $\mathsf{placeT}(P)$. For each transition `r`, we first get the term pattern s that can match the left-hand side of the rule inducing `r` and `placeT` at the same time and then show the commutativity. Mathematically, we show that $s \xrightarrow{\alpha} s_1 \xrightarrow{\mathsf{placeT}(P)} s'$ and $s \xrightarrow{\mathsf{placeT}(P)} s_2 \xrightarrow{\alpha} s'$ for some s_1, s_2, and s'. The rule conditions of `placeT` and the rule inducing `r` are assumed to be met in s. Also, by this lemma's premise, (P, pr, es, dl) is not the top element after inserting in *pQueue*. We prove the independence for each transition.

When $\alpha = \texttt{lookup}$, α and $\texttt{placeT}(P)$ operates on strictly different subsets of K configurations. The term pattern s is acquired by joining two rule's left-hand side (top-side in K rules). $s = \langle\langle x \curvearrowright ...\rangle_{k}\ \langle P'\rangle_{\mathit{id}}\ \langle ...\ x \mapsto l\ ...\rangle_{\mathit{env}}\rangle_{\mathit{program}}\ \langle ...\ l \mapsto v\ ...\rangle_{\mathit{store}}\ \langle t\rangle_{\mathit{time}}\ \langle pq\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$. Then, $s_1 = \langle\langle v \curvearrowright ...\rangle_{k}\ \langle P'\rangle_{\mathit{id}}\ \langle ...\ x \mapsto l\ ...\rangle_{\mathit{env}}\rangle_{\mathit{program}}\ \langle ...\ l \mapsto v\ ...\rangle_{\mathit{store}}\ \langle t\rangle_{\mathit{time}}\ \langle pq\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$ and $s_2 = \langle\langle x \curvearrowright ...\rangle_{k}\ \langle P'\rangle_{\mathit{id}}\ \langle ...\ x \mapsto l\ ...\rangle_{\mathit{env}}\rangle_{\mathit{program}}\ \langle ...\ l \mapsto v\ ...\rangle_{\mathit{store}}\ \langle t\rangle_{\mathit{time}}\ \langle \mathit{insert}(pq, (P, pr, es, dl))\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$, and $s' = \langle\langle x \curvearrowright ...\rangle_{k}\ \langle P'\rangle_{\mathit{id}}\ \langle ...\ x \mapsto l\ ...\rangle_{\mathit{env}}\rangle_{\mathit{program}}\ \langle ...\ l \mapsto v\ ...\rangle_{\mathit{store}}\ \langle t\rangle_{\mathit{time}}\ \langle \mathit{insert}(pq, (P, pr, es, dl))\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$.

When $\alpha = \texttt{assign}, \texttt{if-true}, \texttt{if-false}, \texttt{fbcall}, \texttt{fbExec}, \texttt{fbQuit}$, similar to the case of $\texttt{lookup}$, α and $\texttt{placeT}$ work on no mutual K configuration. Thus, the proof method is identical to the one of $\texttt{lookup}$.

When $\alpha = \texttt{execute}(P)$, $s = \langle\cdot\rangle_{\mathit{active}}\ \langle(P', \_, es'\, dl')\ ...\rangle_{\mathit{pQueue}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\, \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$. $s_1 = \langle P'\rangle_{\mathit{active}}\ \langle(P', \_, es'\, dl')\ ...\rangle_{\mathit{pQueue}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle ...\rangle_{k}\rangle_{\mathit{program}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$. Since, (P, pr, es, dl) cannot be on top of pQueue after insertion by the lemma's premise, $s_2 = \langle\cdot\rangle_{\mathit{active}}\ \langle(P', \_, es'\, dl')\ ...\rangle_{\mathit{pQueue}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$. Finally, $s' = \langle P'\rangle_{\mathit{active}}\ \langle(P', \_, es'\, dl')\ ...\rangle_{\mathit{pQueue}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle ...\rangle_{k}\rangle_{\mathit{program}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ ...$.

When $\alpha = \texttt{endProgram}(P)$, $s = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto \iota\ ...\rangle_{\mathit{interval}}\ \langle P'\rangle_{\mathit{active}}\ \langle Q_1\ (P', pr', es', dl')\ Q_2\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\, \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P\rangle_{\mathit{id}}\ \langle\cdot\rangle_{k}\rangle_{\mathit{program}}\ ...$. Then, $s_1 = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto \iota\ ...\rangle_{\mathit{interval}}\ \langle\cdot\rangle_{\mathit{active}}\ \langle Q_1\ Q_2\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\ (P', pr', es' + \iota, dl' + \iota)\rangle_{\mathit{futureTS}}\ \langle\langle P\rangle_{\mathit{id}}\ \langle\cdot\rangle_{k}\rangle_{\mathit{program}}\ ...$. $s_2 = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto \iota\ ...\rangle_{\mathit{interval}}\ \langle P'\rangle_{\mathit{active}}\ \langle \mathit{insert}(Q_1\ (P', pr', es', dl')\ Q_2, (P, pr, es, dl))\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P\rangle_{\mathit{id}}\ \langle\cdot\rangle_{k}\rangle_{\mathit{program}}\ ...$. $s' = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto \iota\ ...\rangle_{\mathit{interval}}\ \langle\cdot\rangle_{\mathit{active}}\ \langle \mathit{insert}(Q_1\, Q_2,\ (P, pr, es, dl))\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\ (P', pr', es'{+}\iota, dl'{+}\iota)\rangle_{\mathit{futureTS}}\ \langle\langle P\rangle_{\mathit{id}}\ \langle\cdot\rangle_{k}\rangle_{\mathit{program}}\ ...$.

When $\alpha = \texttt{preempt}(P)$, $s = \langle P''\rangle_{\mathit{active}}\ \langle(P', pr', es', dl')\ ...(P'', pr'', es'', dl'')\ ...\ \rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ \langle\langle P''\rangle_{\mathit{id}}\ \langle \cdot \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ ...$. $s_1 = \langle P'\rangle_{\mathit{active}}\ \langle(P', pr', es', dl')\ ...(P'', pr'', es'', dl'')\ ...\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ (P, pr, es, dl)\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle ...\rangle_{k}\rangle_{\mathit{program}}\ \langle\langle P''\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ ...$. By the lemma premise, pQueue's top element is fixed to (P', pr', es', dl') in s_2. $s_2 = \langle P''\rangle_{\mathit{active}}\ \langle(P', pr', es', dl')\ ...(P'', pr'', es'', dl'')\ ...\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ \langle\langle P''\rangle_{\mathit{id}}\ \langle \cdot \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ ...$. $s' = \langle P'\rangle_{\mathit{active}}\ \langle(P', pr', es', dl')\ ...\ (P'', pr'', es'', dl'')\ ...\rangle_{\mathit{pQueue}}\ \langle \mathit{ft_1}\ \mathit{ft_2}\rangle_{\mathit{futureTS}}\ \langle\langle P'\rangle_{\mathit{id}}\ \langle ...\rangle_{k}\rangle_{\mathit{program}}\ \langle\langle P''\rangle_{\mathit{id}}\ \langle \blacksquare \curvearrowright ...\rangle_{k}\rangle_{\mathit{program}}\ ...$.

(2) The overall proof method is the same with (1). The only cell that $\texttt{tick}$ updates is *time* and the side condition is affected by the content of *pQueue* and *futureTS*. The following rules make no update on *pQueue* and *futureTS*: $\texttt{lookup}$, $\texttt{assign}$, $\texttt{if-true}$, $\texttt{if-false}$, $\texttt{fbcall}$, $\texttt{fbExec}$, $\texttt{fbQuit}$, $\texttt{execute}$, and $\texttt{preempt}$. For these rules, the simple joining method can be employed as we show independence between $\texttt{lookup}$ and $\texttt{placeT}$ in (1). All that remains is $\texttt{placeT}$ and $\texttt{endProgram}$. $\texttt{placeT}$ is excluded by our premise.

When $\alpha = \texttt{endProgram}(P)$, $s = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto \iota\ ...\rangle_{\mathit{interval}}\ \langle P\rangle_{\mathit{active}}\ \langle Q_1\ (P, pr, es, dl)\ Q_2\rangle_{\mathit{pQueue}}\ \langle \mathit{ft}\rangle_{\mathit{futureTS}}\ \langle\langle P\rangle_{\mathit{id}}\ \langle\cdot\rangle_{k}\rangle_{\mathit{program}}\ ...$. $s_1 = \langle t\rangle_{\mathit{time}}\ \langle ...\ P \mapsto$

$\iota\ ...\rangle_{interval}\ \langle\cdot\rangle_{active}\ \langle Q_1\, Q_2\rangle_{pQueue}\ \langle ft\,(P, pr, es+\iota, dl+\iota)\rangle_{futureTS}\ \langle\langle P\rangle_{id}\ \langle\blacksquare \curvearrowright ...\rangle_{k}\rangle_{program}\ ....$ $s_2 = \langle t'\rangle_{time}\ \langle ...\ P' \mapsto \iota\ ...\rangle_{interval}\ \langle P\rangle_{active}\ \langle Q_1\,(P, pr, es, dl)\ Q_2\rangle_{pQueue}\ \langle ft\rangle_{futureTS}\ \langle\langle P\rangle_{id}\ \langle\cdot\rangle_{k}\rangle_{program}\ ...$, where $t' < minDL(Q_1\,(P, pr, es, dl)\ Q_2, ft)$. $s' = \langle t'\rangle_{time}\ \langle ...\ P \mapsto \iota\ ...\rangle_{interval}\langle\cdot\rangle_{active}\ \langle Q_1\, Q_2\rangle_{pQueue}\ \langle ft\,(P, pr, es+\iota, dl+\iota)\rangle_{futureTS}\ \langle\langle P\rangle_{id}\ \langle\blacksquare \curvearrowright\ ...\rangle_{k}\rangle_{program}\ ....$ □

The following theorem states that *ample* in Definition 2 satisfies the ample set conditions regarding atomic propositions that do not modify *time*, *pQueue*, and *futureTS*.

Theorem 2. *For any execution path without continuous infinite application of* `tick`*, ample satisfies the four conditions for partial order reduction.*

Proof. The four conditions are (i) $ample(s) \neq \emptyset$ iff $enabled(s) \neq \emptyset$; (ii) For $s \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$, if α depends on $ample(s)$, $\beta_i \in ample(s)$ for some $i \leq n$.; (iii) if s is not fully expanded, all transitions in $ample(s)$ are invisible; and (iv) any cycle in the reduced state space $\hat{\mathcal{S}}$ contains at least one fully expanded state.

To prove condition (i), suppose $ample(s) \neq \emptyset$, then there are two cases. First, if $\mathsf{placeT}(P) \in enabled(s)$ and the result of $\mathsf{placeT}(P)$ does not change the top element of *pQueue*, then $ample(s) = \{\mathsf{placeT}(P), \mathsf{tick}(\tau)\}$ and by the premise $\mathsf{placeT}(P) \in enabled(s)$. Thus, $enabled(s) \neq \emptyset$ holds. In the other case, $ample(s) = enabled(s)$, thus $enabled(s) \neq \emptyset$. Suppose $ample(s) = \emptyset$. This is clearly not the case when $\mathsf{placeT}(P) \in enabled(s)$ and the result of $\mathsf{placeT}(P)$ does not change the top element of *pQueue*, since if so $ample(s)$ is not empty. Thus, it must be the case where $ample(s) = enabled(s)$ and so $enabled(s) = \emptyset$ holds.

If $\mathsf{placeT}(P) \in enabled(s)$ and the result of $\mathsf{placeT}(P)$ does not change the top element of *pQueue*, then $ample(s) = \{\mathsf{placeT}(P), \mathsf{tick}(\tau)\}$. Lemma 2 shows $\mathsf{placeT}(P)$ is only dependent on $\mathsf{tick}(\tau)$ since it does not change the top element of *pQueue*. Likewise, it is shown in Lemma 2 that $\mathsf{tick}(\tau)$ is independent of all other actions except $\mathsf{placeT}(P)$. Therefore, condition (ii) vacuously holds since transitions in $ample(s)$ are only dependent on other transitions in *ample*. Similarly, in the other case when $ample(s) = enabled(s)$, condition (ii) vacuously holds.

When it is not fully expanded, it is the case when $\mathsf{placeT}(P) \in enabled(s)$ and the result of $\mathsf{placeT}(P)$ does not change the top element of *pQueue*, and $ample(s) = \{\mathsf{placeT}(P), \mathsf{tick}(\tau)\}$. These two transitions are invisible since $\mathsf{tick}(\tau)$ only modifies *time* cell and $\mathsf{placeT}(P)$ only modifies *pQueue* and *futureTS* cells. These cells do not affect any atomic propositions of interest by our assumption. Thus, condition (iii) holds.

Suppose there is a cycle that only consists of states that are not fully expanded. Then, for any state s in the cycle, $ample(s) = \{\mathsf{placeT}(P), \mathsf{tick}(\tau)\}$ when $\mathsf{placeT}(P) \in enabled(s)$ and the result of $\mathsf{placeT}(P)$ does not change the top element of *pQueue*, because this is the only case when s is not fully expanded. However, once $\mathsf{placeT}(P)$ occurs, $(P, \dots)$ are no longer in the *futureTS* cell, so $\mathsf{placeT}(P)$ is not enabled in the next state and the next state must be fully expanded. Thus, the cycle must only apply `tick`(τ) rules indefinitely, which is also not possible because of our assumption. Thus, condition (iv) holds. □

6.2 Internal Transitions without Memory Update

Certain scenarios may be equivalent even if they are not addressed by our ample set approach. Consider the following state: $s = \langle P \rangle_{active}\ \langle \langle x \curvearrowright ... \rangle_k\ \langle P \rangle_{id} \rangle_{program}$ $\langle (P', 1, 0, 30)\ (P, 2, 0, 20) \rangle_{pQueue}$ .... Both `lookup` and `preempt` are enabled in s. Applying `preempt` results in $\langle P' \rangle_{active}\ \langle \langle \blacksquare \curvearrowright x \curvearrowright ... \rangle_k\ \langle P \rangle_{id} \rangle_{program}\ \langle (P', 1, 0, 30)$ $(P, 2, 0, 20) \rangle_{pQueue}$ ..., where `lookup` is not enabled anymore. This makes `lookup` dependent on `preempt` and cannot satisfy the condition (*ii*) of ample set. All rules such as `if-T`, `if-F`, and `fbCall`, which operate internally in a k cell of a program without modifying the memory state show the same phenomena. We call these rules *internal rules*.

Figure 9 shows a state space diagram when an internal transition τ and `preempt` is possible. Each circle represents a state and the active task is shown below each circle. We start from the bottom-left state. Whether we choose τ or `preempt`, it converges to the same state in the top-right state. States within the same dashed oval are indistinguishable from outside because τ does not modify any part of the memory in the system. Therefore, we only need to explore the top path by prioritizing internal rules over `preempt`.

Suppose a state labeling function satisfies the following condition: for any two states s and s' with the same *store* cell, $L(s) = L(s')$. Based on the above observation, we have the following theorem.

Theorem 3. *Consider a state s such that* $\texttt{preempt}(Q), \tau \in$ *enabled*(s)*, where τ is internal. For any* $s \xrightarrow{\texttt{preempt}(Q)} s_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\tau} t$*, there exists* $s \xrightarrow{\tau} s' \xrightarrow{\texttt{preempt}(Q)} s'_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} t$*, such that* $L(s_i) = L(s'_i)$ *for* $1 \le i \le n$*.*

Proof. Once `preempt`(P) is applied, it switches context to program Q from program P. Since τ is an internal transition inside program P, it is not enabled until `execute`(P) is applied. i.e. $\alpha_{n-1} = \texttt{execute}(P)$. Given a path $s \xrightarrow{\texttt{preempt}(Q)} s_1 \xrightarrow{\alpha_1} ... \xrightarrow{\texttt{execute}(P)} s_n \xrightarrow{\tau} t$, actions $\alpha_1, \dots, \alpha_{n-1}$ are applied globally or in program Q. Thus, they are independent from τ. [3] Thus, an execution path $s \xrightarrow{\tau} s' \xrightarrow{\texttt{preempt}[Q]} s'_1 \xrightarrow{\alpha_1} ... \xrightarrow{\texttt{execute}(P)} t$ is possible. Also, since both `preempt`(Q) and τ do not change any part of the `store`, $L(s_i) = L(s'_i)$ for $1 \le i \le n$ by our assumption.

□

[3] This independence can be shown with the same method as Lemma 2.

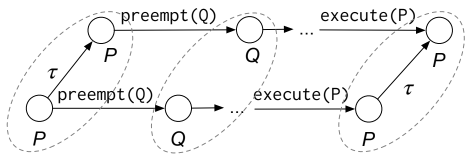

Fig. 9. State diagram when an internal transition and `preempt` are available

7 Experimental Evaluation

To evaluate the effectiveness of our techniques, we have implemented our semantics and state space reduction methods in Maude [8].[4] We have conducted experiments to measure the performance of state space exploration up to a given model time bound. We first compare time and the number of states before and after the abstraction. To emphasize the strength of the abstracted semantics, we also compare it with the sampling-based approach, which is very fast but skips a significant part of the full state space. Then, we compare the time and state space with and without each of the reduction methods in the abstract setting. The experimental results are available at `https://github.com/postechsv/plc-release/releases/tag/v1.0`.

We consider seven models, each with 2 priority settings and 3 model time bounds. The first model is the self-balancing robot in Section 3. We manually adapted and translated the original source's C program into PLC ST programs. The second model is the traffic light example from [20] adapted to a multitask setting. It is a traffic light system for cars and pedestrians coming from all four directions. There are four light controllers, two for cars and two for pedestrians. There is one task for a timer, thus there are up to 5 programs in total.

The LOC of the robot model is 75 and the LOC of the traffic light model is 259. The third to the sixth model is a variant of the second model with different numbers of traffic lights. The seventh benchmark model [5] is from PLCOpen safety library with a significant code size of 2154 lines. It was originally a single-task model, but manually modified to a multitask model.

In the case of concrete semantics, since the nondeterministic `tick` rule is not executable per se, we symbolically execute this semantics using the approach in [20]. In the sampling-based semantics, we increase the time by the greatest common divisor of the intervals of the programs. The sampling method samples the time of the greatest common divisor of the intervals of all the programs. All experiments were conducted on Intel Xeon 2.8 GHz with 256 GB memory. Timeout is set to 1 hour in all settings.

Figure 10 shows the analysis time comparison in scale between concrete and abstract (left) and sampling and abstract (right). The timed-out data is

[4] There are various tools available for executing and analyzing programming languages using K, such as the K tool [19] and Maude [8,31]. The K tool is widely used to perform deductive verification, but the tool is not easily adaptable to model checking with state space reduction techniques.

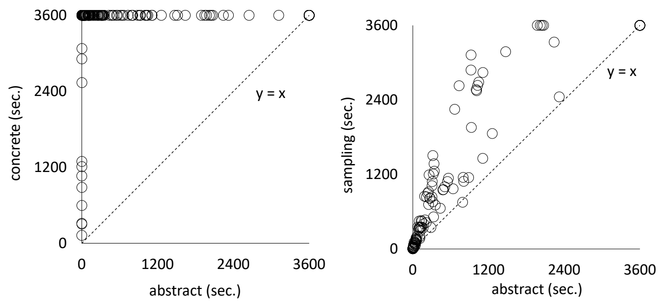

Fig. 10. Analysis time comparison between concrete, sampling, and abstract semantics

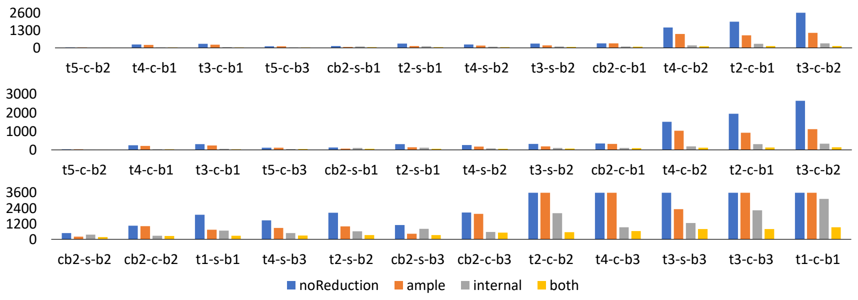

Fig. 11. Analysis time comparison with and without reduction techniques

marked at the edge of the graph. In all cases, state space exploration with time abstraction takes less state space and time than the concrete semantics by far, so the markers are leaning towards the left-hand side. In cases where the execution with concrete semantics is not timed out, the abstract semantics takes at most a hundredth of time. Except for only one case, abstract semantics outperforms the sampling-based semantics.

Figure 11 shows the state space exploration time comparison of abstract semantics with and without each reduction technique. 'noReduction' is the result without any reduction methods, 'ample' with our ample set approach, 'internal' with the reduction using internal memory, and 'both' with both reduction methods. The x-axis shows the benchmark models with their settings. 'r' means our robot models and 't1'-'t5' are the traffic light models with varying numbers of traffic lights. 'cb2' is the one from PLCOpen safety library. 's' and 'c' respectively note the priority setting with fewer possible preemptions (simple) and with greater preemption points (complex). 'b1'-'b3' shows the model's time bound. 'bn' maps to the bound of the greatest common divisor of intervals $\times n$. The y-axis shows the analysis time in seconds. Both methods proved their effectiveness in reducing the state space and analysis times. The internal rule reduction is effective throughout all the settings. The ample set approach is more effective in settings with more equal-priority programs. Applying both reductions proved to be the most efficient. The full tables with detailed information on the benchmark models can be found in Appendix A.

8 Related work

Numerous methods exist for formally analyzing PLC programs written in various languages, including Function Block Diagram [21,26], Sequential Function Chart [18,14,2], Ladder Diagram [28,22], Instruction List [6], and Structured Text [12,10,5]. Most of these approaches utilize a model checking methodology. As mentioned in Section 1, they typically involve translating PLC programs into models that are compatible with existing model checking tools.

The K framework, along with its methodology for semantic definition [29], has been successfully applied to a variety of programming languages, including C [11,15], Java [4], JavaScript [25], Ethereum Virtual Machine [16], etc. In particular, several studies [32,17,20] propose a K semantics for PLC ST. However, preemptive multitasking features and their state-space reduction methods are not considered in these previous K semantics for PLC ST.

A relatively small number of studies deal with multitask PLC. In [13], a technique for symbolic execution of multitask PLC with preemption is presented. However, it is aimed at generating test inputs rather than formal analysis. Another paper [3] focuses on the verification of multitask PLCs with preemption. It is used to verify a specific class of timed multitask PLC program with input delay using the UPAAL tool. However, [3] focuses on Sequential Function Chart (SFC) and Ladder Diagram (LD), whereas our work focuses on ST.

Real-Time Maude [24] provides several formal analysis methods for real-time systems, along with time-complete abstraction [33]. It is based on the *maximal time elapse* strategy, where time elapses until the earliest time at which any event is enabled. However, the maximal time elapse strategy is not complete for multitask PLC ST, because events may happen in arbitrary time. In contrast, our time-optimal semantics is equivalent to the time-complete semantics.

9 Concluding Remarks

We have presented an executable semantics of multitask PLC ST with preemption, based on the K framework. Our semantics efficiently and faithfully covers all possible interleaving scenarios by nondeterministic preemption. We have defined a time-complete semantics that explicitly considers a dense time domain. We have then defined a time abstraction to identify equivalent behaviors across time intervals, resulting in behaviorally equivalent time-abstract semantics.

To cope with the state explosion problem by nondeterministic preemptive multitasking, we have proposed state space reduction techniques based on partial order reduction. We have evaluated the effectiveness of our techniques using several multitask PLC ST benchmarks. The experimental results have shown a significant improvement in the performance of state space exploration using our time-abstract semantics and state space reduction techniques.

There are several limitations to be addressed in the future work. First, our current implementation lacks tool support and we plan to integrate our framework with the tool named STbmc [20]. Second, our semantics do not yet support multi-PLC configurations, where multiple PLCs coordinate to manage and control different parts of a complex process. By exploiting the modularity and extensibility of K semantics, we plan to expand our semantics to support multi-PLC. Lastly, we will apply our method to real-world PLC software to validate its effectiveness in practical scenarios.

Acknowledgement. This work was supported in part by the National Research Foundation of Korea (NRF) grants funded by the Korea government (MSIT) (No. 2021R1A5A1021944 and No. RS-2023-00251577), and by the NATO Science for Peace and Security Programme project SymSafe (grant number G6133).

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking, vol. 26202649. MIT Press (2008)
2. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC Programs Given as Sequential Function Charts, pp. 517–540. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-27863-4_28
3. Bel Mokadem, H., Bérard, B., Gourcuff, V., De Smet, O., Roussel, J.M.: Verification of a timed multitask system with Uppaal. IEEE Transactions on Automation Science and Engineering **7**(4), 921–932 (2010). https://doi.org/10.1109/TASE.2010.2050199
4. Bogdanas, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
5. Bohlender, D., Hamm, D., Kowalewski, S.: Cycle-bounded model checking of PLC software via dynamic large-block encoding. In: Proceedings of the ACM Symposium on Applied Computing. pp. 1891–1898. ACM (2018). https://doi.org/10.1145/3167132.3167334
6. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in Instruction List. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics. vol. 4, pp. 2449–2454. IEEE (2000). https://doi.org/10.1109/ICSMC.2000.884359
7. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. MIT Press (2018)
8. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude manual (version 3.4). Tech. rep., SRI International, Menlo Park (2024)
9. Commission, I.E.: Programmable controllers-part 3: Programming languages. IEC 61131-3 (1993)
10. Darvas, D., Fernández Adiego, B., Blanco Viñuela, E.: PLCverif: A tool to verify PLC programs based on model checking techniques. In: Proceedings of the International

Conference on Accelerator and Large Experimental Physics Control Systems (2015). https://doi.org/10.18429/JACoW-ICALEPCS2015-WEPGF092

11. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. vol. 47, pp. 533–544. ACM (2012). https://doi.org/10.1145/2103656.2103719
12. Gourcuff, V., De Smet, O., Faure, J.M.: Efficient representation for formal verification of PLC programs. In: International Workshop on Discrete Event Systems. pp. 182–187. IEEE (2006). https://doi.org/10.1109/WODES.2006.1678428
13. Guo, S., Wu, M., Wang, C.: Symbolic execution of programmable logic controller code. In: Proceedings of the Joint Meeting on Foundations of Software Engineering. p. 326–336. ACM (2017). https://doi.org/10.1145/3106237.3106245
14. Hassapis, G., Kotini, I., Doulgeri, Z.: Validation of a SFC software specification by using hybrid automata. IFAC Proceedings Volumes **31**(15), 107–112 (1998). https://doi.org/10.1016/S1474-6670(17)40537-4
15. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. vol. 50, pp. 336–345. ACM (2015). https://doi.org/10.1145/2737924.2737979
16. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A complete formal semantics of the Ethereum Virtual Machine. In: Proceedings of IEEE Computer Security Foundations Symposium. pp. 204–217. IEEE (2018). https://doi.org/10.1109/CSF.2018.00022
17. Huang, Y., Bu, X., Zhu, G., Ye, X., Zhu, X., Shi, J.: KST: Executable formal semantics of IEC 61131-3 Structured Text for verification. IEEE Access **7**, 14593–14602 (2019). https://doi.org/10.1109/ACCESS.2019.2894026
18. Lampérière-Couffin, S., Lesage, J.J.: Formal Verification of the Sequential Part of PLC Programs, pp. 247–254. Springer US (2000). https://doi.org/10.1007/978-1-4615-4493-7_25
19. Lazar, D., Arusoaie, A., ŞerbănuŢă, T.F., Ellison, C., Mereuta, R., Lucanu, D., Roşu, G.: Executing formal semantics with the K tool. In: Proceedings of the International Symposium on Formal Methods. Lecture Notes in Computer Science, vol. 7436, pp. 267–271. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_23
20. Lee, J., Kim, S., Bae, K.: Bounded model checking of PLC ST programs using rewriting modulo SMT. In: Proceedings of the ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems. p. 56–67. ACM (2022). https://doi.org/0.1145/3563822.3568016
21. Li, J., Qeriqi, A., Steffen, M., Yu, I.C.: Automatic translation from FBD-PLC-programs to NuSMV for model checking safety-critical control systems. In: Proceedings of the Norsk Informatikkonferanse. Bibsys Open Journal Systems, Norway (2016), `https://dblp.org/rec/conf/nik/LiQSY16.html`
22. Lobov, A., Lastra, J.L.M., Tuokko, R., Vyatkin, V.: Modelling and verification of plc-based systems programmed with ladder diagrams. IFAC Proceedings Volumes **37**(4), 183–188 (2004)
23. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
24. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-order and symbolic computation **20**, 161–196 (2007)

25. Park, D., Stefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991
26. Pavlovic, O., Ehrich, H.D.: Model checking PLC software written in Function Block Diagram. In: Proceedings of the International Conference on Software Testing, Verification and Validation. pp. 439–448. IEEE (2010). https://doi.org/10.1109/ICST.2010.10
27. Peled, D.: Handbook of Model Checking, pp. 173–190. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-10575-8
28. Rausch, M., Krogh, B.H.: Formal verification of PLC programs. In: Proceedings of the American Control Conference. vol. 1, pp. 234–238. IEEE (1998). https://doi.org/10.1109/ACC.1998.694666
29. Rosu, G., Serbănută, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
30. Roşu, G., Şerbănuţă, T.F.: K overview and simple case study. Electronic Notes in Theoretical Computer Science **304**, 3–56 (2014). https://doi.org/10.1016/j.entcs.2014.05.002
31. Şerbănuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: International Workshop on Rewriting Logic and its Applications. pp. 104–122. Springer (2010). https://doi.org/10.1007/978-3-642-16310-4_8
32. Wang, K., Wang, J., Poskitt, C.M., Chen, X., Sun, J., Cheng, P.: K-ST: A formal executable semantics of the Structured Text language for plcs. IEEE Transactions on Software Engineering **49**(10), 4796–4813 (2023). https://doi.org/10.1109/TSE.2023.3315292
33. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. Proceedings of the International Workshop on Rewriting Logic and its Applications **176**(4), 5–27 (2007). https://doi.org/10.1016/j.entcs.2007.06.005

A Experiment

Table 1 shows the analysis time and number of explored states with all the reduction methods. The top row shows the three semantic versions. 'con' and 'abs' respectively map to the semantics before and after abstraction. 'sampling' denotes the sampling-based semantics. The left-most column shows the benchmark model identifiers. The identifiers are the same with Figure 11. 'time(s)' and '#states' show the analysis time in seconds and the number of states explored. 'TO' means timeout.

In all cases, analyses with time abstraction take less state space and time than the concrete semantics by far. In cases where the execution with concrete semantics is not timed out, the abstract semantics takes at most a hundredth of analysis time. Except for only one case, abstract semantics outperforms the sampling-based semantics.

Table 2 shows the analysis time of our abstract semantics and number of explored states with and without ample set approach and internal memory reduction. On the leftmost column, it shows the benchmark models as explained

Table 1. Analysis time and state space size of different semantics with all reduction methods

semantics	con		sampling		abs	
Name	time (s)	#state	time (s)	#state	time (s)	#state
cb2-c-b1	TO	-	312.292	8492	93.636	2537
cb2-c-b2	TO	-	715.54	19397	250.672	6793
cb2-c-b3	TO	-	1001.18	27280	507.276	13897
cb2-s-b1	TO	-	163.96	4425	54.196	1443
cb2-s-b2	TO	-	337.468	9099	162.772	4401
cb2-s-b3	TO	-	515.76	13941	329.68	8969
r-c-b1	313.964	37790	1.753	390	0.684	130
r-c-b2	TO	-	6.204	1346	1.74	358
r-c-b3	TO	-	9.124	1997	3.22	674
r-s-b1	123.196	15206	1.8	372	0.68	134
r-s-b2	1209.912	138886	3.22	677	0.996	199
r-s-b3	TO	-	3.9	827	1.64	338
t1-c-b1	TO	-	2880.212	255727	922.532	82743
t1-c-b2	TO	-	TO	-	TO	-
t1-c-b3	TO	-	TO	-	TO	-
t1-s-b1	TO	-	817.952	78049	273.352	26232
t1-s-b2	TO	-	TO	-	1639.428	151191
t1-s-b3	TO	-	TO	-	TO	-
t2-c-b1	TO	-	347.872	37864	126.92	13484
t2-c-b2	TO	-	1084.264	115957	550.164	58465
t2-c-b3	TO	-	TO	-	TO	-
t2-s-b1	TO	-	144.2	16122	54.36	6067
t2-s-b2	TO	-	759.452	83592	316.912	34729
t2-s-b3	TO	-	6093.6	-	TO	-
t3-c-b1	TO	-	106.18	13723	33.812	4309
t3-c-b2	TO	-	352.616	45171	142.112	18020
t3-c-b3	TO	-	1151.992	144208	799.736	98846
t3-s-b1	TO	-	30.448	4086	11.66	1533
t3-s-b2	TO	-	145.708	19344	63.828	8377
t3-s-b3	TO	-	751.5	96947	790.24	99804
t4-c-b1	TO	-	88.492	11765	27.612	3638
t4-c-b2	TO	-	290.164	38420	117.92	15409
t4-c-b3	TO	-	968.232	127105	639.688	82954
t4-s-b1	TO	-	27.744	3771	11.608	1560
t4-s-b2	TO	-	122.432	16627	54.872	7371
t4-s-b3	TO	-	344.74	46540	288.02	38121
t5-c-b1	2537.608	244998	8.008	1381	2.936	495
t5-c-b2	TO	-	26.816	4637	12.196	2077
t5-c-b3	TO	-	54.732	9617	38.708	6747
t5-s-b1	302.844	32650	1.964	346	0.908	156
t5-s-b2	2911.948	255162	4.576	809	1.728	298
t5-s-b3	TO	-	8.608	1538	6.172	1082

in Table 1. The top row shows the reduction setting. 'nored' is the result without any reduction methods, 'ample' with our ample set approach, 'internal' with the reduction using internal memory, and 'both' with both reduction methods. Both methods proved their effectiveness in reducing the state space and analysis times. The internal memory reduction is effective throughout all the settings. The ample set approach is more effective in settings with a greater number of equal-priority programs. Applying both reductions proved to be the most efficient.

Table 2. Analysis time and state space size of abstract semantics with different combinations of reduction methods

Reduction	noRed		ample		internal		both	
Name	time (s)	#state	time (s)	#state	time (s)	#state	time (s)	#state
cb2-c-b1	341.616	6902	325.64	6592	105.62	2847	93.636	2537
cb2-c-b2	1045.364	20707	1013.14	20397	270.632	7300	250.672	6793
cb2-c-b3	2073.16	40795	1974.464	39489	566.148	15400	507.276	13897
cb2-s-b1	133.808	2676	68.816	1640	99.936	2676	54.196	1443
cb2-s-b2	484.432	9700	212.552	4818	357.696	9700	162.772	4401
cb2-s-b3	1109.632	21376	436.192	9620	807.276	21376	329.68	8969
r-c-b1	3.124	482	2.016	326	1.22	228	0.684	130
r-c-b2	10.54	1652	6.768	1079	3.372	677	1.74	358
r-c-b3	21.296	3396	11.972	1970	6.832	1416	3.22	674
r-s-b1	1.54	232	0.808	134	1.224	232	0.68	134
r-s-b2	4.52	681	1.228	214	3.388	681	0.996	199
r-s-b3	9.488	1438	2.06	353	6.984	1438	1.64	338
t1-c-b1	TO	-	TO	-	3118.364	265005	922.532	82743
t1-c-b2	TO	-	TO	-	TO	-	TO	-
t1-c-b3	TO	-	TO	-	TO	-	TO	-
t1-s-b1	1896.08	119624	735.024	51612	665.092	60076	273.352	26232
t1-s-b2	TO	-	TO	-	-	-	1639.428	151191
t1-s-b3	TO	-	TO	-	TO	-	TO	-
t2-c-b1	1947.76	145593	923.652	73328	304.204	31841	126.92	13484
t2-c-b2	TO	-	TO	-	2021.58	205201	550.164	58465
t2-c-b3	TO	-	TO	-	TO	-	TO	-
t2-s-b1	309.616	23857	140.992	11751	114.888	12143	54.36	6067
t2-s-b2	2048.456	151857	1005.06	81129	613.628	64659	316.912	34729
t2-s-b3	TO	-	TO	-	TO	-	TO	-
t3-c-b1	305.536	29132	235.656	22410	54.02	6918	33.812	4309
t3-c-b2	2649.576	243603	1112.292	104103	336.892	42983	142.112	18020
t3-c-b3	TO	-	TO	-	2241.104	269674	799.736	98846
t3-s-b1	52.552	4999	29.224	2921	20.064	2587	11.66	1533
t3-s-b2	317.896	29689	192.952	18909	101.484	13037	63.828	8377
t3-s-b3	TO	323470	2322.612	216558	1260.06	155005	790.24	99804
t4-c-b1	250.696	24284	216.86	21246	37.12	4876	27.612	3638
t4-c-b2	1515.624	146079	1027.728	99507	187.468	24383	117.92	15409
t4-c-b3	TO	-	TO	-	929.464	119477	639.688	82954
t4-s-b1	49.276	4814	31.728	3212	17.816	2336	11.608	1560
t4-s-b2	258.708	24739	174.796	17391	80.312	10531	54.872	7371
t4-s-b3	1472.68	137564	885.552	87915	476.82	61544	288.02	38121
t5-c-b1	7.988	1029	7.796	1007	3.072	517	2.936	495
t5-c-b2	36.744	4743	36.088	4675	12.596	2145	12.196	2077
t5-c-b3	115.288	15469	110.652	14874	42.072	7342	38.708	6747
t5-s-b1	1.66	213	1.168	156	1.252	213	0.908	156
t5-s-b2	4.2	542	2.228	321	3.16	542	1.728	298
t5-s-b3	17.704	2273	8.06	1105	13.084	2273	6.172	1082

Both reduction techniques prove effective in concrete and sampling semantics as well. Table 3 and Table 4 respectively shows the result for concrete semantics and sampling semantics. The table format is the same with Table 2.

Table 3. Analysis time and state space size of concrete semantics with different combinations of reduction methods

Reduction	noRed		ample		internal		both	
Name	time (s)	#state	time (s)	#state	time (s)	#state	time (s)	#state
cb2-c-b1	TO	-	TO	-	TO	-	TO	-
cb2-c-b2	TO	-	TO	-	TO	-	TO	-
cb2-c-b3	TO	-	TO	-	TO	-	TO	-
cb2-s-b1	TO	-	TO	-	TO	-	TO	-
cb2-s-b2	TO	-	TO	-	TO	-	TO	-
cb2-s-b3	TO	-	TO	-	TO	-	TO	-
r-c-b1	TO	-	TO	-	882.056	101868	313.964	37790
r-c-b2	TO	-	TO	-	TO	-	TO	-
r-c-b3	TO	-	TO	-	TO	-	TO	-
r-s-b1	TO	-	1063.44	87406	596.236	68978	123.196	15206
r-s-b2	TO	-	TO	-	TO	-	1209.912	138886
r-s-b3	TO	-	TO	-	TO	-	TO	-
t1-c-b1	TO	-	TO	-	TO	-	TO	-
t1-c-b2	TO	-	TO	-	TO	-	TO	-
t1-c-b3	TO	-	TO	-	TO	-	TO	-
t1-s-b1	TO	-	TO	-	TO	-	TO	-
t1-s-b2	TO	-	TO	-	TO	-	TO	-
t1-s-b3	TO	-	TO	-	TO	-	TO	-
t2-c-b1	TO	-	TO	-	TO	-	TO	-
t2-c-b2	TO	-	TO	-	TO	-	TO	-
t2-c-b3	TO	-	TO	-	TO	-	TO	-
t2-s-b1	TO	-	TO	-	TO	-	TO	-
t2-s-b2	TO	-	TO	-	TO	-	TO	-
t2-s-b3	TO	-	TO	-	TO	-	TO	-
t3-c-b1	TO	-	TO	-	TO	-	TO	-
t3-c-b2	TO	-	TO	-	TO	-	TO	-
t3-c-b3	TO	-	TO	-	TO	-	TO	-
t3-s-b1	TO	-	TO	-	TO	-	TO	-
t3-s-b2	TO	-	TO	-	TO	-	TO	-
t3-s-b3	TO	-	TO	-	TO	-	TO	-
t4-c-b1	TO	-	TO	-	TO	-	TO	-
t4-c-b2	TO	-	TO	-	TO	-	TO	-
t4-c-b3	TO	-	TO	-	TO	-	TO	-
t4-s-b1	TO	-	TO	-	TO	-	TO	-
t4-s-b2	TO	-	TO	-	TO	-	TO	-
t4-s-b3	TO	-	TO	-	TO	-	TO	-
t5-c-b1	TO	-	TO	-	3075.4	128504	2537.608	244998
t5-c-b2	TO	-	TO	-	TO	-	TO	-
t5-c-b3	TO	-	TO	-	TO	-	TO	-
t5-s-b1	TO	-	TO	-	1291.444	128504	302.844	32650
t5-s-b2	TO	-	TO	-	TO	-	2911.948	255162
t5-s-b3	TO	-	TO	-	TO	-	TO	-

Table 4. Analysis time and state space size of sampling semantics with different combinations of reduction methods

Reduction	noRed		ample		internal		both	
Name	time (s)	#state	time (s)	#state	time (s)	#state	time (s)	#state
cb2-c-b1	1252.208	25556	1207.5	24628	349.512	9420	312.292	8492
cb2-c-b2	2685.436	56792	2547.36	54935	815.48	22042	715.54	19397
cb2-c-b3	-	-	-	-	1138.772	30920	1001.18	27280
cb2-s-b1	456.592	9138	206.256	5213	342.776	9138	163.96	4425
cb2-s-b2	954.952	19082	424.768	10779	706.992	19082	337.468	9099
cb2-s-b3	1458.22	29410	655.368	16569	1088.464	29410	515.76	13941
r-c-b1	9.872	1596	6.632	1117	3.224	639	1.753	390
r-c-b2	63.972	10861	23.708	3968	17.18	3732	6.204	1346
r-c-b3	108.468	18445	36.32	6270	29.188	6480	9.124	1997
r-s-b1	3.908	618	2.076	372	3.108	618	1.8	372
r-s-b2	14.468	2248	3.948	747	10.88	2248	3.22	677
r-s-b3	22.84	3572	4.816	907	17.156	3572	3.9	827
t1-c-b1	TO	-	TO	-	TO	-	2880.212	255727
t1-c-b2	TO	-	TO	-	TO	-	TO	-
t1-c-b3	TO	-	TO	-	TO	-	TO	-
t1-s-b1	TO	-	2627.708	188557	2249.22	200619	817.952	78049
t1-s-b2	TO	-	TO	-	TO	-	TO	-
t1-s-b3	TO	-	TO	-	TO	-	TO	-
t2-c-b1	TO	-	3123.244	251236	838.384	88420	347.872	37864
t2-c-b2	TO	-	TO	-	TO	-	1084.264	115957
t2-c-b3	TO	-	TO	-	TO	-	TO	-
t2-s-b1	1084.032	86998	442.484	37938	340.06	36844	144.2	16122
t2-s-b2	TO	-	TO	-	TO	-	759.452	83592
t2-s-b3	TO	-	TO	-	TO	-	6093.6	-
t3-c-b1	1017.596	98020	895.808	87480	137.636	17804	106.18	13723
t3-c-b2	TO	-	2839.92	269467	1371.808	174324	352.616	45171
t3-c-b3	TO	-	TO	-	3330.288	403469	1151.992	144208
t3-s-b1	178.912	17669	91.528	9404	58.648	7631	30.448	4086
t3-s-b2	1503.56	144634	466.86	46732	449.028	58700	145.708	19344
t3-s-b3	TO	-	2448.628	240441	1858.484	230085	751.5	96947
t4-c-b1	919.272	90900	838.46	83649	110.484	14610	88.492	11765
t4-c-b2	TO	-	2632.932	256644	847.176	110873	290.164	38420
t4-c-b3	TO	-	TO	-	1956.592	252117	968.232	127105
t4-s-b1	154.14	15516	86.3	9020	48.416	6465	27.744	3771
t4-s-b2	1191.74	117435	405.588	41141	354.304	47116	122.432	16627
t4-s-b3	3175.476	304550	1148.844	119740	949.968	123640	344.74	46540
t5-c-b1	25.192	3315	24.556	3254	8.392	1442	8.008	1381
t5-c-b2	107.556	14573	82.344	11069	35.456	6176	26.816	4637
t5-c-b3	198.352	27231	167.368	23086	67.28	11839	54.732	9617
t5-s-b1	3.828	510	2.504	346	2.928	510	1.964	346
t5-s-b2	11.756	1540	5.824	901	8.764	1540	4.576	809
t5-s-b3	21.368	2817	11.104	1644	15.972	2817	8.608	1538