K-RAPID: A Formal Executable Semantics of the RAPID Robot Programming Language

Zichen Wang
College of Control Science and Engineering, Zhejiang University
Hangzhou, China
withnorman@zju.edu.cn

Jingyi Wang*
College of Control Science and Engineering, Zhejiang University
Hangzhou, China
wangjyee@zju.edu.cn

Fu Song
[1]Key Laboratory of System Software (Chinese Academy of Sciences)
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
songfu@ios.ac.cn

Kun Wang
College of Control Science and Engineering, Zhejiang University
Hangzhou, China
kunwang_yml@zju.edu.cn

Hongyi Pu
College of Control Science and Engineering, Zhejiang University
Hangzhou, China
hongyipu.zju@gmail.com

Peng Cheng
College of Control Science and Engineering, Zhejiang University
Hangzhou, China
lunarheart@zju.edu.cn

Abstract

Industrial robots are widely used in industrial production as mechanical devices. It is essential to guarantee that their control software operates safely and properly, as any functional or security-related defects may lead to serious incidents. However, industrial robots are programmed mostly in proprietary languages varying from vendor to vendor, making it challenging to formally analyze their correctness in a unified way. One of the most representative robot programming languages is the RAPID language proposed by ABB Robotics. In this paper, we present K-RAPID, a formal executable semantics of RAPID in the K-Framework ($\mathbb{K}$). K-RAPID is developed according to the official ABB documentation and defined in a generic extensible manner. It can be used either for validating the correctness of compiler implementation or analyzing the control programs written in RAPID. We evaluate the correctness of K-RAPID by executing 563 test programs collected from multiple sources and comparing the results against the official robot simulation environment RobotStudio. The results suggest that K-RAPID covers the core features of RAPID correctly. Moreover, we show how we could apply K-RAPID to verify RAPID programs using LTL model checking and to provide a formal specification of RAPID to uncover inappropriate behaviors in the programs.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Security and privacy → Formal methods and theory of security**.

Keywords: Industrial robots, Formal methods in robotics and automation, Robot programming, RAPID language, K-Framework

ACM Reference Format:
Zichen Wang, Jingyi Wang, Fu Song, Kun Wang, Hongyi Pu, and Peng Cheng. 2024. K-RAPID: A Formal Executable Semantics of the RAPID Robot Programming Language. In *10th ACM Cyber-Physical System Security Workshop (CPSS '24), July 2, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626205.3659149

1 Introduction

With the widespread application of industrial robots, safety incidents caused by the wrong operations of robots occur from time to time [14, 18, 19]. Thus, industrial robots should be ensured to operate safely and properly in order to eliminate the threat that abnormal operations pose to human and equipment safety.

The operating logic of an industrial robot is determined by its control program, which is pre-programmed by robot operators with the aid of some proprietary domain-specific languages [11]. A robot programming language contains a series of instructions that supervise the robot's activities including moving the robots, manipulating end effectors, and communicating with robot operators. These functionalities make robot control programs security- and safety-critical. Any functional or security-related defects (e.g. setting the

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CPSS '24, July 2, 2024, Singapore, Singapore

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0420-8/24/07
https://doi.org/10.1145/3626205.3659149

destination of a move instruction to an unreachable position) may lead to harmful consequences. Hence, *it is critical to guarantee that control programs for industrial robots are correctly implemented and exhibit expected behaviors when they are executed.*

Several works attempt to look into the security and safety risks related to robot software. Pogliani et al. [16] propose a static code analyzer for ABB's and KUKA's robot control programs, and discover insecure or potentially malicious issues in the use of sensitive primitives. Mandal et al. [12] develop a static program analysis framework for RAPID using the .NET framework and further extend it to apply to PLC programming languages [13]. Wang et al. [22] use model checking to rigorously verify the correctness of code generation models of ROS systems. Existing works are limited since they lack the aid of formal methods. To balance the accuracy and efficiency of static analysis and model checking, it is necessary to fully comprehend the purpose and structure of each robot control program. Such firm understanding is inseparable from a complete formal semantics.

To the best of our knowledge, there has been no work on the formal semantics of industrial robot programming languages so far. Compared to previous work on mainstream languages like C [5] and Java [3], several new challenges need to be addressed for the formal semantics of robot programming languages. First, robot programming languages have a bunch of features to control the operations of robots. These features are not supported by general-purpose languages. Thus, formal semantics of general-purpose programming languages cannot be directly applied to robot languages. Besides, robot programming languages are proprietary languages varying from vendor to vendor, making it challenging to define a generic formal semantics available for diverse robot programming language implementations.

To tackle the challenges, we propose the first formal semantics of a widely used specific language called RAPID. All the control programs of ABB's robots are written in RAPID [4] , which has complex program structures and complete instruction functions. We formalize the semantics in the K-Framework [17] and name it as K-RAPID. We evaluate the correctness of K-RAPID by executing 563 test programs and comparing the results against the robot simulation environment RobotStudio [2]. We find that K-RAPID can run 529 of these programs correctly. We also evaluate the completeness of K-RAPID by analyzing the types of features that it covers. It turns out that K-RAPID has covered the core features of RAPID, showing its high completeness. K-RAPID also supports LTL model checking on RAPID programs, offering functionalities besides just interpreting RAPID programs. As a demonstration, we present a case study that performs model checking on the control program of an industrial sorting system and illustrates the ability of model checking to find program flaws. Furthermore, K-RAPID provides a formal specification of RAPID and we find two categories of inappropriate behaviors in RAPID with the aid of K-RAPID.

2 Preliminaries

2.1 Overview of Robot Programming

An industrial robot, as shown in Figure 1(a), is an "*automatically controlled, reprogrammable, multipurpose manipulator programmable in three or more axes, which can be either fixed in place or mobile for use in industrial automation applications*" as defined in ISO 8373 [9]. They are often deployed in relatively harsh environments to complete complex production activities. We abstract the architectural design of an industrial robot into the form illustrated in Figure 1(b) for easy understanding. Specifically, an industrial robot is a mechanical arm with multiple joints, controlled by a *control unit* and terminated by an *end effector* (e.g., welding gun, gripper, cutter) that interacts with the environment.

The control unit plays a crucial role in the operations of industrial robots. It is a complex device consisting of collaborative hardware and software systems. The control unit determines the operating logic of the manipulator and drives its movement by means of control programs, which are preprogrammed by robot operators. These programs can be developed with the aid of the FlexPendant or the Integrated Development Environment (IDE). Programming on the FlexPendant is often used when the operator is on the production site and the control program is relatively simple, while the IDE is suitable for building more complex programs remotely. The IDE (e.g. ABB's RobotStudio), which is aided by simulators, allows programmers to write and debug control programs in a safe offline environment.

There are no standards for robot programming languages, instead, leading robot manufacturers design their own languages. RAPID [15] is one of the most representative robot programming languages. It is a high-level programming language for industrial robots of ABB Robotics with complex program structures, rich instruction functions and operands, to control the robot, set the output, read the input, etc.

We introduce the entire process of robot programming by analyzing the following RAPID program. It moves the end effector to the point `p0` and displays a message on the FlexPendant.

```
MODULE MainModule
  PERS robtarget p0 := [...];
  PERS tooldata grip := [...];
  PROC main()
      MoveJ p0, v500, fine, grip;
      TPWrite "Successfully moved to p0!";
  ENDPROC
ENDMODULE
```

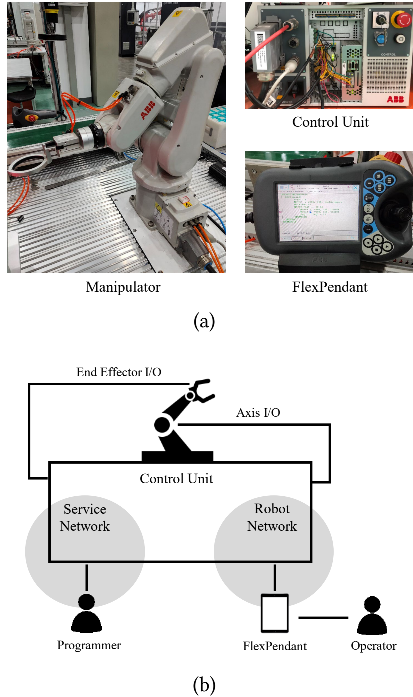

Figure 1. (a) An industrial robot and (b) its architecture overview.

More specifically, it first defines two persistent variables `p0` and `grip`, where `p0` of type `robtarget` denotes a destination point, and `grip` of type `tooldata` contains parameters of the end effector, such as load mass and center of gravity position. In the `main` procedure, the controller drives the end effector to the point `p0` via the `MoveJ` Instruction. Note that the predefined parameters `v500` and `fine` specify the moving speed and positioning level of the end effector. Next, the program displays a string on the screen of the FlexPendant via the `TPWrite` instruction to prompt the operator that the move operation is finished.

2.2 The K-Framework

The K-Framework ($\mathbb{K}$) provides a way to define rewrite-based executable semantics of programming languages using configurations and rules [17]. It has been used to formalize a few mainstream programming languages, e.g., C [8], Java [3], Python [7], PHP [6], Rust [20] and Solidity [10].

The architecture of the K-Framework is illustrated in Figure 2. The formal semantics, which is the core of $\mathbb{K}$, consists of two parts, namely the runtime configuration and the semantic rules. Note that formal semantics describe the behavior of each statement in a programming language and need to be manually defined with reference to the language's technical manual. The semantic rules parse the programs statement by statement, during which they exchange the runtime states with the runtime configuration. The formal semantics is compiled into the interpreter or model checker

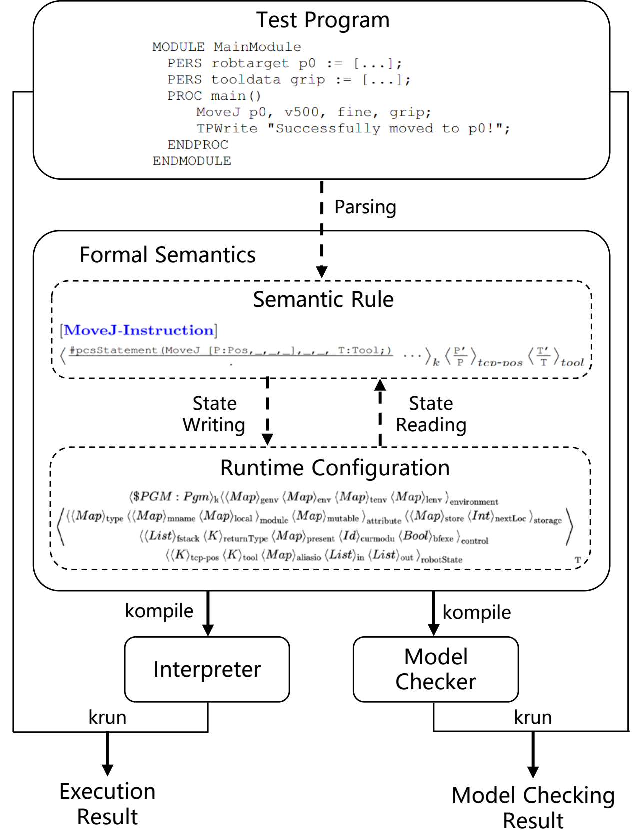

Figure 2. Overview of the K-Framework

of the programming language (utilizing the `kompile` command), which is used to execute or verify the target program (utilizing the `krun` command).

3 K-RAPID: The Formal Semantics of RAPID in $\mathbb{K}$

3.1 Runtime Configuration

The RAPID programs need to update several kinds of states when being executed in $\mathbb{K}$. To monitor these states well, we need to delicately design the structure of the runtime configuration. We show the configuration defined in K-RAPID in Figure 3.

Overview. In this configuration, there are six main cells in the whole configuration cell `T` named `k`, `environment`, `attribute`, `storage`, `control` and `robotState`. Each cell contains one or more sub-cells, the values of which are initialized in the configuration with its type specified.

Program execution. In the cell `k`, a source program named `Pgm` is stored for execution. If the program stored in cell `k` is terminated in a normal manner, the content of the cell will become a dot, indicating that the cell is empty and there are no more programs to execute.

Execution environment. During the execution of RAPID programs, we need to save the runtime environment of the program, i.e., the location of variables, constants, functions, procedures, and other objects in the storage, involving the cell `environment` and the cell `control`. The cell `environment` consists of `genv`, `env`, `tenv` and `lenv`, which record the mappings from objects to their locations. The cell `control`, consisting of `fstack`, `returnType`, `present`, `curmodu` and `bfexe`, is used to store the intermediate states during execution.

Memory operation. The memory operation involves the cell `storage` and the cell `attribute`. In the cell `storage`, there are two cells named `store` and `nextLoc`. The cell `store` records the values of all the defined variables and constants and stores the return types, the arguments, and the bodies of the defined routines. When a new variable or a new routine is declared, a new location denoted by an integer will be allocated from the `nextLoc` cell. As a result, the integer value in `nextLoc` will be increased by one. The cell `local` records whether the variables and the routines have the attribute `LOCAL` or not.

Robot states. Since the RAPID language covers the features of robot movement, IO, etc., we need to use a cell to record the relevant states, which involves five cells named `tcp-pos`, `tool`, `aliasio`, `in` and `out`. The `tcp-pos` cell is used to indicate the position information of the robot's tool center point (TCP). The `tool` cell records the tool data currently being used by the robot. To improve the portability of the program, the I/O signals used in the RAPID program can be named freely. When the program is used for a specific robot, the program I/O and the actual system configuration I/O can be bounded by defining the I/O signal with an alias name. The cell `aliasio` is used to record such alias names of I/O signals. The communication between the robot controller and the FlexPendant is the most commonly used communication operation of the RAPID programs. The user can input information to the controller through the FlexPendant. The controller can also output information to the screen of the FlexPendant. Here, we use `in` and `out` to simulate the FlexPendant.

3.2 Semantic Rules

Next, we introduce the executable operational semantic rules of RAPID formalized in the K-Framework. We mainly introduce semantic rules that are not common in other languages or are challenging to describe in operational semantics.

Module Declaration. A RAPID program consists of one or more program modules. Global data declarations and routine definitions are within the modules. Before executing or analyzing the programs, $\mathbb{K}$ needs to parse the modules in the programs one by one, complete the data definition work at the beginning of each module, and save the definition of each routine for subsequent calls.

`RULE` (1) specifies how the runtime configuration is affected after the execution of the module declaration statement. The statement, which is located in the cell `k`, mainly consists of three parts: module name `M`, a list of module data declarations `Ds`, and a list of routine declarations `Rs`. We rewrite the statement by using a sequence of function `#pcsModDataDeclarations` and function `#pcsRoutDeclarations`. The symbol $\curvearrowright$ here stands for "followed by". Specifically, `#pcsModDataDeclarations(M,Ds)` deals with each data declaration in the module. `#pcsRoutDeclarations(M,Rs)` is used to deal with routine declarations.

Routine Declaration and Routine Calls. In RAPID, routines can be divided into user routines and predefined routines. A user routine is defined by a RAPID routine declaration, while a predefined routine is supplied by the system. We consider two types of routines, which are procedures and functions. The difference between a function and a procedure is whether there is a return value in its definition.

1) User Routines: `RULE` (2) shows how functions are declared by the K-Framework. Declaring a RAPID function is achieved through `FUNC T:Type F:Id(Ps:Paras) B:RoutBody ENDFUNC`, where `T` is the type of the return value, `F` is the function name, `Ps` specifies the parameters of the function, and `B` is the function body. First, we add a mapping of `F` and `L` in the cell `genv`. The function is not local, so we record `false` in `local`. `mname` record the name of the module to which the function belongs. In the cell `store`, we store `M`, `T`, `Ps` and `B` in `#lambda` for subsequent calls.

When a user function is called, the function name will be first transformed into a `#lambda` function. Then we can proceed with the call according to `RULE` (3). In the cell `k`, we rewrite the function call as three sub-steps, which are the `#mkDecls`, the `#pcsRoutBody`, and the `RETURN` statement. To be specific, `#mkDecls` passes the values of the arguments of the function call to the parameters, `#pcsRoutBody` decomposes the function body, and the `RETURN` statement results in a forced return. Meanwhile, the local environment is updated to `GEnv`, and we back up the former environment in `tenv`. The current module is changed to `M`, to which the function belongs. We also update the type of return value in `returnType` and clear `present`. In addition, the current environment should be pushed to `fstack`.

The semantic rules of procedure declaration and procedure call are basically the same as the function, except that there are no return values.

2) Predefined Routines: 78 predefined routines are covered by K-RAPID in total. Since the predefined routines are officially provided by the ABB, such routines are not declared in user programs. Thus, we do not need to define the semantic rules for declarations of the predefined routines. As for predefined routine calls, since we cannot obtain their implementation details, each routine has its unique semantic rule. We summarize the list of covered predefined functions in Table 2 for space reasons.

$$\left\langle \begin{array}{c} \langle \$PGM : Pgm \rangle_{\mathrm{k}} \langle \langle Map \rangle_{\mathrm{genv}} \, \langle Map \rangle_{\mathrm{env}} \, \langle Map \rangle_{\mathrm{tenv}} \, \langle Map \rangle_{\mathrm{lenv}} \, \rangle_{\mathrm{environment}} \\ \langle \langle Map \rangle_{\mathrm{type}} \, \langle \langle Map \rangle_{\mathrm{mname}} \, \langle Map \rangle_{\mathrm{local}} \, \rangle_{\mathrm{module}} \, \langle Map \rangle_{\mathrm{mutable}} \, \rangle_{\mathrm{attribute}} \, \langle \langle Map \rangle_{\mathrm{store}} \, \langle Int \rangle_{\mathrm{nextLoc}} \, \rangle_{\mathrm{storage}} \\ \langle \langle List \rangle_{\mathrm{fstack}} \, \langle K \rangle_{\mathrm{returnType}} \, \langle Map \rangle_{\mathrm{present}} \, \langle Id \rangle_{\mathrm{curmodu}} \, \langle Bool \rangle_{\mathrm{bfexe}} \, \rangle_{\mathrm{control}} \\ \langle \langle K \rangle_{\mathrm{tcp\text{-}pos}} \, \langle K \rangle_{\mathrm{tool}} \, \langle Map \rangle_{\mathrm{aliasio}} \, \langle List \rangle_{\mathrm{in}} \, \langle List \rangle_{\mathrm{out}} \, \rangle_{\mathrm{robotState}} \end{array} \right\rangle_{\mathrm{T}}$$

Figure 3. The $\mathbb{K}$ configuration for the states of RAPID programs

Variable Declaration and Assignment. Next, we present the semantic rules for declarations and assignments of variables. Variables declared in the module can be either global or local. A global variable can be accessed not only inside the module to which it belongs but also outside the module.

`RULE` (4) specifies the semantic rule of global variable declaration in a RAPID module. In the cell `k`, a variable of type `T`, named `X`, is declared using the function `#pcsModDataDeclaration`. The variable belongs to the module `M` and its value is initialized during declaration. The initialization operation is rewritten by the function `#initialize`. A mapping from the variable name and its location in the storage is recorded in `genv` and `env`. Similar to `RULE` (2), the cell `local`, `type`, `mname`, `mutable` and `nextLoc` are updated accordingly.

The semantic rule (5) is defined for the ordinary assignment, such as assigning a new value to a variable of type `num`. It says that the assignment operation could be rewritten to a `#write` (`RULE` (6)) function only if `X` is mutable.

Then we come to the read operation on variables and routines shown in `RULE` (7). Here, we consider the object `X` as a variable. First, we need to obtain the location `L` of `X` in `env`. According to `L`, we can get the value `V` in `store`. Note that `RULE` (7) could be executed only if a prerequisite is satisfied. The prerequisite requires that `X` is not local, or the program is before the execution phase, or `X` is local meanwhile it's being accessed inside the module.

Move Instructions. In addition to common functionalities, one of the important features of RAPID is that it is specially designed to control robots. More importantly, there are instructions for making a robot move. K-RAPID covers 9 move instructions in total and we introduce one of them, `MoveJ`, due to limited space. The semantic rule of the `MoveJ` instruction is specified in `RULE` (8). As we can see, the `MoveJ` instruction has 4 parameters. We mainly concentrate on the first and the fourth. The first parameter, which is of `robtarget` type, identifies the target TCP of the robot's movement. The target TCP consists of four elements, and the first one among them is a three-dimensional coordinate data `P` of `Pos` type. The fourth parameter in the instruction is the tool data `T` of the tool used by the robot. The robot movements are programmed as pose-to-pose movements, i.e. "move from the current position to a new position". The path between these two positions is then automatically calculated by the robot. Unfortunately, we cannot model the robot's trajectory through the K-Framework because the end effector's trajectory planning algorithm is unavailable. Thus, the trajectory of the robot cannot be recorded in the runtime configuration. What we can do is update the target position data from `P'` to `P` in the cell `tcp-pos`, and update the tool data from `T'` to `T` in the cell `tool`.

I/O Operations. The robot can be equipped with I/O signals that can be read and changed in the user program. The signals are used for communication with external equipment the robot cooperates with. Input signals are set by the external equipment and can be used in the RAPID program to initiate the operation of the robot. Output signals are set by the RAPID program to signal that the external equipment should do something.

`RULE` (9) specifies the semantic rule of AliasIO instruction, which bounds an I/O unit with its alias name. Specifically, given an I/O unit named after `X1`, we get its location information in the cell `env`. Then we add a mapping from `L` to the alias name `X2` of the unit in the cell `aliasio`.

The value of a digital signal is whether `0` or `1`. Some instructions can change the values of the digital I/O signals in RAPID. We introduce one of them, which is `Set` instruction being used to set the value of an output signal to `1`. The usage of the `Set` instruction is divided into two situations. The first one is that the signal name `X` is not an alias name. Then the signal is directly sent to `1` by using `RULE` (10).

User Interaction. In the ABB robots, messages can be transferred between the controller and the FlexPendant. There are several instructions for sending information to the robot operator or receiving input from the operator. In K-RAPID, we simulate the FlexPendant by using the console. Instead of getting user input from the FlexPendant, the program receives input from the standard input. Similarly, user output is displayed through the standard output. We introduce two instructions `TPWrite` and `TPReadNum`.

`TPWrite` outputs information to the FlexPendant. As shown in `RULE` (11), a string `S` with a line break is put to the cell `out`, which represents the standard output.

`TPReadNum` reads a number from the FlexPendant. `RULE` (12) specifies the semantic rule of `TPReadNum`. The format of the instruction is `TPReadNum X:Id,S:String;`, in which `X` is the name of the variable storing the input number and `S` is a string that will be output to the FlexPendant. `S` is output through another `TPWrite` instruction. After that, the input number is read through a `#tpreadnum` function, the semantic rule of which is `RULE` (13). An input number is read in the cell `in` and is assigned to variable `X`.

Table 1. The partial semantic rules of K-RAPID

Module Declaration

[Module-Declaration] (1)

$$\left\langle \frac{\texttt{MODULE M:Id Ds:ModDataDeclarations Rs:RoutDeclarations ENDMODULE}}{\texttt{\#pcsModDataDeclarations(M,Ds)} \curvearrowright \texttt{\#pcsRoutDeclarations(M,Rs)}} \cdots \right\rangle_k$$

Routine Declaration and Routine Calls

[Function-Declaration] (2)

$$\left\langle \frac{\texttt{\#pcsRoutDeclaration(M:Id,FUNC T:Type F:Id(Ps:Paras) B:RoutBody ENDFUNC)}}{\cdot} \cdots \right\rangle_k \left\langle \frac{\texttt{GEnv}}{\texttt{GEnv[F<-L]}} \right\rangle_{genv} \left\langle \frac{\texttt{L}}{\texttt{L +Int 1}} \right\rangle_{nextLoc}$$

$$\left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->false}} \cdots \right\rangle_{local} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->M}} \cdots \right\rangle_{mname} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->\#lambda(M,T,Ps,B)}} \cdots \right\rangle_{store}$$

[User-Function-Call] (3)

$$\left\langle \frac{\texttt{\#lambda(M:Id,T:Type,Ps:Paras, B:RoutBody)(As:Args)} \curvearrowright \texttt{K}}{\texttt{\#mkDecls(Ps,As)} \curvearrowright \texttt{\#pcsRoutBody(B)} \curvearrowright \texttt{RETURN;}} \right\rangle_k \langle \texttt{GEnv} \rangle_{genv} \left\langle \frac{\texttt{Env}}{\texttt{GEnv}} \right\rangle_{env} \left\langle \frac{\texttt{TEnv}}{\texttt{Env}} \right\rangle_{tenv} \left\langle \frac{\texttt{M'}}{\texttt{M}} \right\rangle_{curmodu} \left\langle \frac{\texttt{T'}}{\texttt{T}} \right\rangle_{returnType}$$

$$\left\langle \frac{\texttt{P'}}{\texttt{.List}} \right\rangle_{present} \left\langle \frac{\texttt{.List}}{\texttt{ListItem((T',Env,TEnv,M',P',K))}} \cdots \right\rangle_{fstack}$$

Variable Declaration and Assignment

[Module-Data-Declaration] (4)

$$\left\langle \frac{\texttt{\#pcsModDataDeclaration(M:Id, VAR T:Type X:Id:=E:Expression;)}}{\texttt{\#initialize(L,E,T)}} \cdots \right\rangle_k \left\langle \frac{\texttt{GEnv}}{\texttt{GEnv[X<-L]}} \right\rangle_{genv} \left\langle \frac{\texttt{Env}}{\texttt{Env[X<-L]}} \right\rangle_{env} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->false}} \cdots \right\rangle_{local}$$

$$\left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->T}} \cdots \right\rangle_{type} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->M}} \cdots \right\rangle_{mname} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->true}} \cdots \right\rangle_{mutable} \left\langle \frac{\texttt{L}}{\texttt{L +Int 1}} \right\rangle_{nextLoc}$$

[Assignment] (5)

$$\left\langle \frac{\texttt{\#pcsStatement(X:Id:=E:Expression;)}}{\texttt{\#write(L,E,T)}} \cdots \right\rangle_k \langle \cdots \texttt{X|->L} \cdots \rangle_{env} \langle \cdots \texttt{L|->T} \cdots \rangle_{type} \langle \cdots \texttt{L|->true} \cdots \rangle_{mutable}$$

[Write] (6)

$$\left\langle \frac{\texttt{\#write(L:Int,V:Value,T:Type)}}{\cdot} \cdots \right\rangle_k \left\langle \cdots \frac{\texttt{L|->V'}}{\texttt{L|->V}} \cdots \right\rangle_{store} \quad \textbf{requires } \texttt{\#equalType(T,\#typeof(V)) ==K true}$$

[Read] (7)

$$\left\langle \frac{\texttt{X:Id}}{\texttt{V}} \cdots \right\rangle_k \langle \cdots \texttt{X|->L} \cdots \rangle_{env} \langle \cdots \texttt{L|->B1} \cdots \rangle_{local} \langle \texttt{M2} \rangle_{curmodu} \langle \texttt{B2} \rangle_{bfexe} \left\langle \cdots \frac{\texttt{L}}{\texttt{V:Value}} \cdots \right\rangle_{store} \langle \cdots \texttt{L|->M1} \cdots \rangle_{mname}$$

$$\textbf{requires } \texttt{B1 ==K false orBool (M1 ==K M2 andBool B1 ==K true) orBool B2 ==K true}$$

Move Instructions

[MoveJ-Instruction] (8)

$$\left\langle \frac{\texttt{\#pcsStatement(MoveJ [P:Pos,_,_,_],_,_, T:Tool;)}}{\cdot} \cdots \right\rangle_k \left\langle \frac{\texttt{P'}}{\texttt{P}} \right\rangle_{tcp\text{-}pos} \left\langle \frac{\texttt{T'}}{\texttt{T}} \right\rangle_{tool}$$

I/O Operations

[AliasIO-Instruction] (9)

$$\left\langle \frac{\texttt{\#pcsStatement(AliasIO X1:Id,X2:Id;)}}{\cdot} \cdots \right\rangle_k \langle \cdots \texttt{X2|->L} \cdots \rangle_{env} \left\langle \cdots \frac{\texttt{.Map}}{\texttt{L|->X1}} \cdots \right\rangle_{aliasio}$$

[Set-Instruction] (10)

$$\left\langle \frac{\texttt{\#pcsStatement(Set X:Id;)}}{\cdot} \cdots \right\rangle_k \langle M \rangle_{aliasio} \langle \cdots \texttt{X|->L} \cdots \rangle_{env} \langle \cdots \texttt{L|->signaldo} \cdots \rangle_{type} \langle \cdots \texttt{L|->true} \cdots \rangle_{mutable}$$

$$\left\langle \cdots \frac{\texttt{L|->_}}{\texttt{L|->1}} \cdots \right\rangle_{store} \quad \textbf{requires } \texttt{L in_keys(M) ==K false}$$

User Interaction

[TPWrite-Instruction] (11)

$$\left\langle \frac{\texttt{\#pcsStatement(TPWrite S:String;)}}{\cdot} \cdots \right\rangle_k \left\langle \cdots \frac{\texttt{.List}}{\texttt{ListItem(S +String "\textbackslash{} n")}} \right\rangle_{out}$$

[TPReadNum-Instruction] (12)

$$\left\langle \frac{\texttt{\#pcsStatement(TPReadNum X:Id,S:String;);}}{\texttt{\#pcsStatement(TPWrite S;)} \curvearrowright \texttt{\#tpreadnum(X)}} \cdots \right\rangle_k$$

[TPReadNum-Auxiliary-Function] (13)

$$\left\langle \frac{\texttt{\#tpreadnum(X:Id)}}{\texttt{\#pcsStatement(X:=I;)}} \cdots \right\rangle_k \left\langle \frac{\texttt{ListItem(I:Int)}}{\texttt{.List}} \cdots \right\rangle_{in}$$

4 Semantics Evaluation

In this section, we elaborate on how to systematically evaluate K-RAPID. We implement it in the K-Framework release 5.1.234 with around 7500 lines of code and compile it using the `kompile` command in $\mathbb{K}$. The experiments that run the test programs by K-RAPID are conducted on a virtual machine configured with Ubuntu 20.04 LTS. The experiments that run the test programs on RobotStudio are conducted on a virtual machine configured with Windows 7.

4.1 Overview of the Test Process

Our overall test workflow, which is displayed in Figure 4, consists of 4 steps: data collecting and preprocessing, program executing, consistency checking, and coverage analysis.

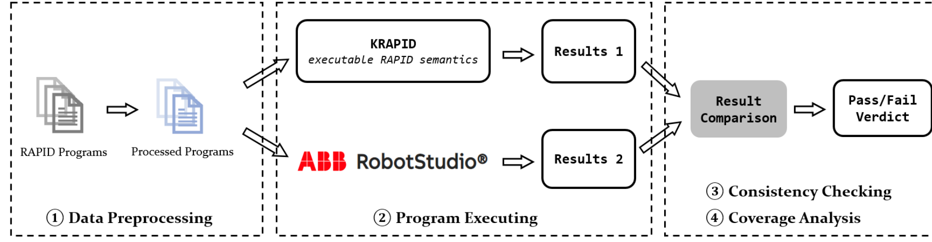

Figure 4. Overview of the test process

Table 2. Predefined routines covered by K-RAPID

Categories		Covered Routines
Arithmetic Operation	Function	Abs(), AbsDnum(), Sqrt(), SqrtDnum(), Exp(), Pow(), PowDnum(), Round(), Trunc(), TruncDnum(),Sin(), RoundDnum(), SinDnum(), Cos(), CosDnum(), Tan(), TanDnum(), ASin(), ASinDnum(), ACos(), ACosDnum(), ATan(), ATanDnum(), ATan2(), ATan2Dnum(), BitAnd(), BitOr(), BitNeg(), BitLSh(), BitRSh(), BitCheck(), StrDigCalc(), StrDigCmp(), NumToDnum(), DnumToNum(), StrPart(), ByteToStr(), StrToByte(), DecToHex(), HexToDec()
	Procedure	Clear, Add, Incr, Decr, TryInt
Move Operation	Function	Offs(), CPos(), CTool(), VectMagn(), Distance(), Dotprod()
	Procedure	MoveJ, MoveC, MoveL, MoveJSync, MoveLSync, MoveCSync, MoveJDO, MoveLDO, MoveCDO
Return and Stop	Procedure	RETURN, EXIT, Break, ExitCycle, Stop
I/O Operation	Function	DInput(), DOutput(), TestDI(), ValidIO()
	Procedure	AliasIO, AliasIOReset, Set, Reset, InvertDO, SetDO
User Interaction	Procedure	TPWrite, TPReadNum, TPReadFK

Data collecting and preprocessing. First, we collect RAPID programs to be tested from multiple sources and preprocess them as test programs. To evaluate K-RAPID as accurately as possible, we construct a test suite consisting of multiple sources summarized in Table 3. We collect 103 publicly available test programs and generate an additional 460 programs based on the mutation algorithm in [21] to increase the diversity of test samples. In total, we test our semantics with 563 test programs covering diverse semantic features. Besides, we need to perform data preprocessing on the collected RAPID programs. To be specific, all the comments are removed from the test programs. We also remove infinite loops to facilitate the observation of the program execution results. Furthermore, we change all the main module names to `MainModule` and the main routine names to `main`, so that K-RAPID can find the entry points of the programs. For programs without a main routine, we manually add one and call other routines in it so that the programs can execute normally.

Program Execution. We utilize the processed RAPID programs as inputs to test the correctness of K-RAPID. The execution procedure is divided into two parts. First, we use K-RAPID to execute the programs and get the results named `Results 1`. In the meantime, we input the programs to RobotStudio, ABB's official simulation and offline programming software, to simulate the operations of the robots when executing the programs. Then we get the execution results named `Results 2`.

Consistency Checking. After finishing the program execution period, we carry out consistency checking on the execution results to evaluate the correctness of K-RAPID. In other words, we manually compare `Results 1` and `Results 2` and get the result of a pass or fail verdict. Details are introduced in subsection 4.2.

Coverage Analysis. We then do coverage analysis by checking out the source code of K-RAPID and the consistency checking results. If all tests involving a particular feature mentioned in K-RAPID pass the checking, we consider that the feature is fully covered. The results are displayed in subsection 4.2.

Table 3. Test program sources and the corresponding file number

Source		Pass/Total
publicly available	Github	39/55
	ABB robot forum	13/13
	Academic papers	3/3
	User manuals of RAPID	9/9
	Example programs from RobotStudio	19/23
Generated by mutation algorithm		446/460
Total		529/563

4.2 Experiment Results and Analyses

As aforementioned, we evaluate K-RAPID from two perspectives, which are semantics correctness and semantics

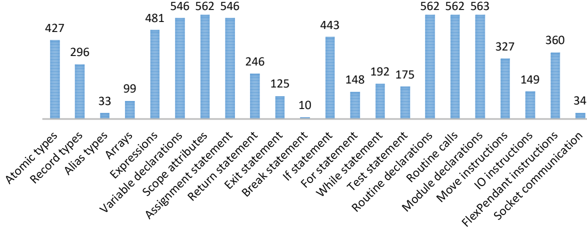

Figure 5. Number of tests for each feature in the test set

completeness. To be specific, the correctness shows the consistency of program execution results between K-RAPID and the robot itself, and the semantics completeness denotes the coverage of K-RAPID. For the sake of safety and convenience, we carry out our experiments on the official ABB robot simulation software RobotStudio instead of ABB robot entities. Our analyses show that the proposed semantics K-RAPID has covered the core features mentioned in the official RAPID documentation [1] and the execution results are consistent with RobotStudio.

Semantics correctness. We test the K-RAPID interpreter on the test cases mentioned above. As described in subsection 4.1, the test programs are separately input to K-RAPID and RobotStudio. The evaluation is carried out by manually comparing the execution behaviors of K-RAPID with the ones of RobotStudio. We regard a test program as correct if the result from K-RAPID is consistent with that from RobotStudio. Referring to the data in Table 3, among the 563 programs, 34 of them cannot be successfully parsed by K-RAPID because they contain socket communication instructions that are not covered by K-RAPID. Besides, the remaining 529 programs all pass the consistency checking.

Semantics completeness. We manually count the number of tests for each important feature among the 563 tests and list the results in Figure 5. As indicated in Figure 5, the atomic type and record type are the most common types in the test set. The assignment statement and the if statement are the most common statements. Besides, expressions, variable declarations, scope attributes, routine declarations, routine calls, and module declarations appear so frequently that they are present in almost every program.

We assume that all programs in the test set that can be parsed by K-RAPID collectively form a set $\mathbb{P}$. If all tests in $\mathbb{P}$ involving a particular feature pass the consistency checking, we consider that the feature is fully covered. The coverage of K-RAPID is listed in Table 4 from the perspective of each feature specified by the official RAPID documentation. Table 4 indicates that **K-RAPID can cover all features listed in Figure 5 except socket communication.** Concerning data types, K-RAPID covers 27 data types in RAPID, which are the most used. These types fall into 3 categories: atomic types, record types, and alias types. Among the three categories, record types are composite types composed of one or more atomic types and alias types are alias of atomic

Table 4. Coverage of the proposed RAPID semantics

Features	Coverage	Features	Coverage
Data Types		`Exit Statement`	✓
`Atomic Types`	✓	`Break Statement`	✓
`Record Types`	✓	`If Statement`	✓
`Alias Types`	✓	`For Statement`	✓
Arrays		`While Statement`	✓
`Linear Arrays`	✓	`Test Statement`	✓
Expressions		`Label Statement`	✓
`Arithmetic Operations`	✓	**Routines**	
`Logical Operations`	✓	Routine Declarations	
`Bitwise Operations`	✓	`Function Declarations`	✓
`Relation Operations`	✓	`Procedure Declarations`	✓
`Look Up`	✓	Routine Calls	
`Index Access`	✓	`Function Calls`	✓
Variable Declarations		`Procedure Calls`	✓
`VAR`	✓	**Modules**	
`PERS`	✓	`Module Declarations`	✓
`CONST`	✓	**Instructions**	
Scope Attributes		`Move Instructions`	✓
`GLOBAL`	✓	`IO Instructions`	✓
`LOCAL`	✓	`FlexPendant Instructions`	✓
Statements		`Socket Communication`	✗
`Assignment Statement`	✓		
`Return Statement`	✓		

✓: Covered ✗: Not Covered

types or record types. For arrays, K-RAPID covers both linear arrays and multi-dimensional arrays. Expressions such as arithmetic operations, logical operations, bitwise operations, relation operations, look-up operations, and index access are covered. In RAPID, there are variable types named `VAR`, `PERS` and `CONST`, all supported by K-RAPID. Furthermore, when declaring variables or routines, scope attributes `GLOBAL` and `LOCAL` should be declared. So they are also included in the proposed semantics. The semantics of statements listed in Figure 5 are all covered. As for routines, we implement declarations and calls for functions and procedures. User-defined routines and predefined routines are covered. Also, module declarations are covered. Lastly, as a key part of controlling the operations of the robots, instructions can not be ignored. K-RAPID has covered the most important instructions which are move instructions, I/O instructions, and FlexPendant instructions. The reason K-RAPID does not support socket communication is that they are used by industrial robots to communicate with servers and other robots, whereas we are currently only focusing on offline single robotic arms. Socket communication with other entities is technically a difficult problem in $\mathbb{K}$, so we leave the formal definitions of socket communication for future work.

5 LTL Model Checking

In this section, we present a real-world case study to illustrate an application of K-RAPID in verifying RAPID programs. Specifically, we use Linear Temporal Logic (LTL) to analyze the properties of a robot control program in an industrial sorting system, which provides a function of sorting defective and non-defective precision pinions.

The workflow of the industrial sorting system is illustrated in Figure 6, and the corresponding control program A is in Section A.1. First, all the pinions, whether defective or not, are shipped on `Conveyor1` in a single direction of movement. Once a pinion reaches the inspection zone `B`, `Conveyor1` stops and the pinion is inspected by the visual sensor `Sensor1` and the gravity sensor `Sensor2`. `Sensor1` detects whether the pinion has surface defects, while `Sensor2` detects whether the weight of the pinion is within the preset range. A pinion that does not pass the inspection of either sensor is considered to be defective. Next, the inspected pinion is sorted by an ABB robot that uses a gripper as the end effector. A defective pinion will be transferred to `Conveyor3` and a non-defective one will be transferred to `Conveyor2`. `Conveyor2` and `Conveyor3` run continuously to transport the pinions to the target location. Six I/O signals are involved in the system:

- `C_IN` is the status signal for the motor of `Conveyor1`. "1" means the belt is working, and "0" means it is stopped.
- `C_OUT` is the control signal that the robot outputs to the motor of `Conveyor1`. "1" means the robot commands `conveyor1` to work, and "0" means the robot commands `Conveyor1` to stop.
- `S_1` is the signal of `Sensor1`. "1" means the pinion is defective, and "0" means it is non-defective.
- `S_2` is the signal of `Sensor2`. "1" means the pinion is defective, and "0" means it is non-defective.
- `S_3` is the signal of `Sensor3`. "1" means the pinion is transferred to `C1`, and "0" means it is not transferred to `C1`.
- `S_4` is the signal of `Sensor4`. "1" means the pinion is transferred to `C2`, and "0" means it is not transferred to `C2`.

Throughout the sorting process, the intent of the robot is: When a non-defective pinion is detected, the robot moves from point `A` to `B` and grips the pinion. Then the robot moves to point `C1` and drops the pinion onto `Conveyor2`. Finally, the robot moves back to `A` and lets `Conveyor1` continue to work. When a defective pinion is detected, the robot behaves similarly, except that the pinion is dropped on `Conveyor3` instead of `Conveyor2`.

We summarize three properties from the sorting system and list the corresponding LTL formulas. *P1*: To ensure that the pinion can be swimmingly inspected by the sensors and gripped by the robot, `Conveyor1` should stop when the pinion reaches the inspecting zone (i.e., `C_IN` is false). When a pinion is identified as non-defective (i.e. `S_1` and `S_2` are both true), the pinion will be transported to `C1` (i.e. `S_3` is true and `S_4` is false). Thus, we can represent *P1* as:

$$\square(\neg \mathtt{C_IN} \wedge (\mathtt{S_1} \wedge \mathtt{S_2}) \rightarrow \Diamond(\mathtt{S_3} \wedge \neg \mathtt{S_4})),$$

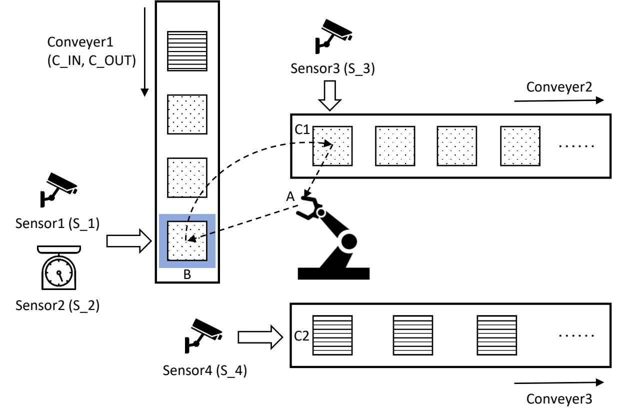

Figure 6. Overview of the sorting system

where ∧ denotes "always", ◊ denotes "eventually" and ¬ denotes "not". *P2*: Similar to *P1*, when a pinion is identified as defective (i.e. `S_1` or `S_2` is false), the pinion will be transported to `C2` (i.e. `S_3` is false and `S_4` is true). Thus, we can represent *P2* as:

$$\square(\neg \mathtt{C_IN} \wedge \neg(\mathtt{S_1} \wedge \mathtt{S_2}) \rightarrow \Diamond(\neg \mathtt{S_3} \wedge \mathtt{S_4})).$$

P3: After the pinion is transported (i.e. either `S_3` is true or `S_4` is true), the robot moves back to the starting point `A` and restart `Conveyor1` (i.e. `C_OUT` is set to 1). Thus, we can represent *P3* as:

$$\square((\mathtt{S_3} \wedge \neg \mathtt{S_4}) \vee (\neg \mathtt{S_3} \wedge \mathtt{S_4}) \rightarrow \Diamond(\mathtt{C_OUT})).$$

A specified RAPID program can be verified with the aforementioned LTL formulas by utilizing the built-in LTL model checking tool in $\mathbb{K}$. The basic idea is to explore the state space of the control program by executing it on the interpreter generated by $\mathbb{K}$ that checks whether the specified properties are violated. The model checker finds a counterexample for the property if there is a violation, or the property is verified to be true. Besides, there are two flawed RAPID programs that show the negligence of the programmers. Specifically, program B (in Section A.2) is basically the same as the non-defective program, but the target transfer position `C2` of the defective pinions is mistakenly set to `C1`. The consequence of this is that all the pinions are transported to `C1`, which violates property `P2`. Program C (in Section A.3) does not set the signal `C_OUT` to 1 after finishing the transport, resulting in `Conveyor1` not continuing to run. The behavior of program C violates property `P3`. Both of the violations can be detected using the model checker.

6 Formal Specification of RAPID

For programmers, the use of programming languages should be as consistent as possible with the intent of the language designers. However, the actual situation is not always satisfactory. The main reference material for RAPID language

Table 5. Inappropriate Behaviors Found by K-RAPID

Categories	Examples
Inconsistencies between documents and RobotStudio	Inconsistent requirements for parameter case in documents and RobotStudio. e.g. `MoveL start, v2000, z40, grip3 \WObj:=fixture` is correct both in the official documentation and RobotStudio; `MoveL start, v2000, z40, grip3 \Wobj:=fixture` is correct in RobotStudio, but wrong in the documentation.
Undefined behaviors in the documentation	RobotStudio restricts the parameter `tooldata` in move instructions to only PERS type, but there is no clear restriction on this in the official documentation. e.g. `MoveL start, v2000, z40, grip3 \WObj:=fixture` is correct regardless of whether `grip3` is of type VAR or of type PERS. However, `grip3` must be of type PERS to be correct in RobotStudio.

programming is the official documentation provided by ABB, and the development environment is RobotStudio. Program developers refer to the official documentation to write and run control programs on RobotStudio. Much of the content in RAPID documents is described in natural language, which can easily lead to ambiguities or undefined behaviors. Also, there exist behaviors of the compiler in RobotStudio inconsistent with official documentation or inconsistent with programming conventions. For these reasons, program developers may write programs with inappropriate behaviors.

K-RAPID offers a formal specification of industrial robot control programs written in the RAPID language. With the assistance of K-RAPID, developers can write correct control programs for ABB robots. In Table 5 we give some examples to illustrate this.

First, we introduce an example of inconsistency between official documents and RobotStudio. Move instructions in RAPID, such as `MoveL` and `MoveC`, contain optional parameter `\WObj` which specifies the data of the work object it is operating on. It is noteworthy that the spelling of `\WObj` is case-sensitive according to RAPID documentation. However, RobotStudio ignores this case-sensitivity and developers can use identifiers such as `\Wobj` to replace `\WObj`. This leads to inconsistency between RAPID official documentation and RobotStudio. Similar problems exist with optional parameters like `\Tool` and `\Num`. Since K-RAPID strictly stipulates the case of identifiers in the syntax and semantic rules, such problems are avoided.

There are undefined behaviors that are processed in RobotStudio but not clearly stated in the documentation. For instance, RobotStudio restricts `tooldata` in move instructions to only PERS type, not VAR or CONST type. Such undefined behaviors can be eliminated with the specification that the formal semantics provides.

7 Conclusion

In this work, we introduce an executable operational semantics K-RAPID for RAPID formalized in the K-Framework. K-RAPID covers the core features of RAPID, such as routine declaration and routine calls, variable declaration and assignment, move instructions, I/O operations, and user interaction. We systematically test K-RAPID by using test programs collected from multiple sources to show its correctness and completeness. Besides, we perform LTL model checking on RAPID programs based on K-RAPID and make it a formal specification to uncover multiple inappropriate behaviors of RAPID, demonstrating that K-RAPID can provide functionalities besides just interpreting RAPID programs.

Acknowledgments

The work was partially supported by Fundamental Research Funds for the Central Universities of China under Grants 226-2024-00048 and partially supported by National Science Foundation of China (NSFC) under Grants 62293511.

References

[1] ABB. 2008. Technical reference manual - RAPID kernel.
[2] ABB. 2022. RobotStudio Suite. https://new.abb.com/products/robotics/robotstudio.
[3] D. Bogdnaş and G. Roşu. 2015. K-Java: A complete semantics of Java. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 2015 (2015), 445–456.
[4] Sudeep Chakravarty. 2019. World's Top 10 Industrial Robot Manufacturers. https://www.marketresearchreports.com/blog/2019/05/08/world%E2%80%99s-top-10-industrial-robot-manufacturers.
[5] Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 533–544.
[6] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014 – Object-Oriented Programming*. 567–592.
[7] Dwight Guth. 2013. A formal semantics of Python 3.3. http://hdl.handle.net/2142/45275.

[8] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the undefinedness of C. In *PLDI 2015 - Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* Association for Computing Machinery, 336–345.

[9] International Organization for Standardization. 2012. International Organization for Standardization, Robots and robotic devices - Vocabulary, ser. International standard; ISO 8373.

[10] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* 1695–1712.

[11] Federico Maggi and Marcello Pogliani. 2020. Rogue Automation - Vulnerable and Malicious Code in Industrial Programming. https://documents.trendmicro.com/assets/white_papers/wp-rogue-automation-vulnerable-and-malicious-code-in-industrial-programming.pdf.

[12] Avijit Mandal, Raoul Jetley, Meenakshi D'Souza, and Sreeja Nair. 2017. A Static Analyzer for Industrial Robotic Applications. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).* 24–27.

[13] Avijit Mandal, Devina Mohan, Raoul Jetley, Sreeja Nair, and Meenakshi D'Souza. 2018. A Generic Static Analysis Framework for Domain-specific Languages. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. 27–34.

[14] Jeff Martin. 2016. Robotic machine kills woman at auto parts plant, two weeks before wedding. https://www.ctvnews.ca/world/robotic-machine-kills-woman-at-auto-parts-plant-two-weeks-before-wedding-1.3204248.

[15] Michal. 2019. What programming language is used for robotics? https://roboticsbook.com/what-programming-language-is-used-for-robotics/.

[16] Marcello Pogliani, Federico Maggi, Marco Balduzzi, Davide Quarta, and Stefano Zanero. 2020. Detecting Insecure Code Patterns in Industrial Robot Programs. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (2020).

[17] Runtime Verification Inc. 2022. K Semantic Framework. https://kframework.org/.

[18] Staff RT. 2015. Factory Robot Kills Worker in India. https://www.roboticsbusiness-review.com/rbr/factory_robot_kills_worker_in_india/.

[19] Staff RT. 2015. Robot Kills Worker in Volkswagen Factory Accident. https://www.roboticsbusinessreview.com/rbr/robot_kills_worker_in_volkswagen_factory_accident/#:~:text=Robot%20Kills%20Worker%20in%20Volkswagen%20Factory%20Accident%20A,injuries.%20By%20RT%20Staff%20%7C%20July%201%2C%202015.

[20] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018.* 44–51.

[21] Kun Wang, Jingyi Wang, Christopher M. Poskitt, Xiangxiang Chen, Jun Sun, and Peng Cheng. 2023. K-ST: A Formal Executable Semantics of the Structured Text Language for PLCs. *IEEE Transactions on Software Engineering* 49, 10 (2023), 4796–4813. https://doi.org/10.1109/TSE.2023.3315292

[22] Rui Wang, Yong Guan, Houbing Song, Xinxin Li, Xiaojuan Li, Zhiping Shi, and Xiaoyu Song. 2019. A Formal Model-Based Design Method for Robotic Systems. *IEEE Systems Journal* 13, 1 (2019), 1096–1107.

A The Control Programs of the Sorting Systems

A.1 Program A

```
MODULE MainModule

    // Define the target positions of robot.
    VAR robtarget a := [...];
    VAR robtarget b := [...];
    VAR robtarget c1 := [...];
    VAR robtarget c2 := [...];

    // Define the speed of the robot.
    VAR speeddata v := [...];

    // Define the zonedata of the robot.
    VAR zonedata z:= [...];

    // Define the end effector of the robot.
    VAR tooldata grip := [...];

    // Declare the I/O signals.
    VAR signaldi s1;
    VAR signaldi s2;
    VAR signaldi s3;
    VAR signaldi s4;
    VAR signaldi c_in;
    VAR signaldo c_out;

    // Define the I/O units.
    CONST string config_cin := “C_IN”;
    CONST string config_cout := “C_OUT”
    CONST string config_s1 := ”S_1”;
    CONST string config_s2 := ”S_2”;
    CONST string config_s1 := ”S_3”;
    CONST string config_s2 := ”S_4”;

    PROC main()
        Bool flag_1 := false;
        Bool flag_2 := false;
        Bool flag_3 := false;
        Bool flag_4 := false;

        // Link the I/O units with the I/O variables.
        AliasIO config_cin, c_in;
        AliasIO config_cout, c_out;
        AliasIO config_s1, s1;
        AliasIO config_s2, s2;
        AliasIO config_s3, s3;
        AliasIO config_s4, s4;
        WHILE true DO
            // Transport the non-defective pinions
            // and restart the conveyor.
            flag_1 := c_in=0 AND s1=1 AND s2=1;
            IF flag_1 THEN
                MoveJ b, v, zone, grip;
                Grip();
                MoveJ c1, v, zone, grip;
```

```
                Drop();
                MoveJ a, v, zone, grip;
                flag_3 = (s3 = 1) AND (s4 = 0);
                IF flag_3 THEN
                    SetDO c_out, 1;
                ENDIF
            ENDIF

            // Transport the defective pinions
            // and restart the conveyor.
            flag_2:=c_in=0 AND NOT s1=1 AND s2=1;
            IF flag_2 THEN
                MoveJ b, v, zone, grip;
                Grip();
                MoveJ c2, v, zone, grip;
                Drop();
                MoveJ a, v, zone, grip;
                flag_4 = (s3 = 0) AND (s4 = 1);
                IF flag_4 THEN
                    SetDO c_out, 1;
                ENDIF
            ENDIF
        ENDWHILE
    ENDPROC

    // Grip the pinions.
    PROC grip()
        ...
    ENDPROC

    // Drop the pinions.
    PROC drop()
        ...
    ENDPROC ENDMODULE
```

A.2 Program B

```
MODULE MainModule

    // Define the target positions of robot.
    VAR robtarget a := [...];
    VAR robtarget b := [...];
    VAR robtarget c1 := [...];
    VAR robtarget c2 := [...];

    // Define the speed of the robot.
    VAR speeddata v := [...];

    // Define the zonedata of the robot.
    VAR zonedata z:= [...];

    // Define the end effector of the robot.
    VAR tooldata grip := [...];

    // Declare the I/O signals.
    VAR signaldi s1;
    VAR signaldi s2;
    VAR signaldi s3;
    VAR signaldi s4;
    VAR signaldi c_in;
    VAR signaldo c_out;

    // Define the I/O units.
    CONST string config_cin := “C_IN”;
    CONST string config_cout := “C_OUT”
    CONST string config_s1 := ”S_1”;
    CONST string config_s2 := ”S_2”;
    CONST string config_s1 := ”S_3”;
    CONST string config_s2 := ”S_4”;

    PROC main()
        Bool flag_1 := false;
        Bool flag_2 := false;
        Bool flag_3 := false;
        Bool flag_4 := false;

        // Link the I/O units with the I/O variables.
        AliasIO config_cin, c_in;
        AliasIO config_cout, c_out;
        AliasIO config_s1, s1;
        AliasIO config_s2, s2;
        AliasIO config_s3, s3;
        AliasIO config_s4, s4;
        WHILE true DO
            // Transport the non-defective pinions
            // and restart the conveyor.
            flag_1 := c_in=0 AND s1=1 AND s2=1;
            IF flag_1 THEN
                MoveJ b, v, zone, grip;
                Grip();
                MoveJ c1, v, zone, grip;
                Drop();
                MoveJ a, v, zone, grip;
                flag_3 = (s3 = 1) AND (s4 = 0);
                IF flag_3 THEN
                    SetDO c_out, 1;
                ENDIF
            ENDIF

            // Transport the defective pinions
            // and restart the conveyor.
            flag_2:=c_in=0 AND NOT s1=1 AND s2=1;
            IF flag_2 THEN
                MoveJ b, v, zone, grip;
                Grip();
                // Mistakenly move the defective
                // pinions to c1.
                MoveJ c1, v, zone, grip;
                Drop();
                MoveJ a, v, zone, grip;
                flag_4 = (s3 = 0) AND (s4 = 1);
                IF flag_4 THEN
                    SetDO c_out, 1;
                ENDIF
            ENDIF
        ENDWHILE
```

```
    ENDPROC

    // Grip the pinions.
    PROC grip()
        ...
    ENDPROC

    // Drop the pinions.
    PROC drop()
        ...
    ENDPROC ENDMODULE
```

A.3 Program C

```
MODULE MainModule

    // Define the target positions of robot.
    VAR robtarget a := [...];
    VAR robtarget b := [...];
    VAR robtarget c1 := [...];
    VAR robtarget c2 := [...];

    // Define the speed of the robot.
    VAR speeddata v := [...];

    // Define the zonedata of the robot.
    VAR zonedata z:= [...];

    // Define the end effector of the robot.
    VAR tooldata grip := [...];

    // Declare the I/O signals.
    VAR signaldi s1;
    VAR signaldi s2;
    VAR signaldi s3;
    VAR signaldi s4;
    VAR signaldi c_in;
    VAR signaldo c_out;

    // Define the I/O units.
    CONST string config_cin := "C_IN";
    CONST string config_cout := "C_OUT"
    CONST string config_s1 := "S_1";
    CONST string config_s2 := "S_2";
    CONST string config_s1 := "S_3";
    CONST string config_s2 := "S_4";

    PROC main()
        Bool flag_1 := false;
        Bool flag_2 := false;

        // Link the I/O units with the I/O variables.
        AliasIO config_cin, c_in;
        AliasIO config_cout, c_out;
        AliasIO config_s1, s1;
        AliasIO config_s2, s2;
        AliasIO config_s3, s3;
        AliasIO config_s4, s4;
        WHILE true DO
            // Forget to restart the conveyor
            // after transporting.
            // Transport the non-defective pinions
            // and restart the conveyor.
            flag_1 := c_in=0 AND s1=1 AND s2=1;
            IF flag_1 THEN
                MoveJ b, v, zone, grip;
                Grip();
                MoveJ c1, v, zone, grip;
                Drop();
                MoveJ a, v, zone, grip;
            ENDIF

            // Transport the defective pinions
            // and restart the conveyor.
            flag_2:=c_in=0 AND NOT s1=1 AND s2=1;
            IF flag_2 THEN
                MoveJ b, v, zone, grip;
                Grip();
                MoveJ c2, v, zone, grip;
                Drop();
                MoveJ a, v, zone, grip;
            ENDIF
        ENDWHILE
    ENDPROC

    // Grip the pinions.
    PROC grip()
        ...
    ENDPROC

    // Drop the pinions.
    PROC drop()
        ...
    ENDPROC ENDMODULE
```