Semantics of a Language for Distributed Coordination and Control

Ritwika Ghosh Sayan Mitra

University of Illinois at Urbana-Champaign
rghosh9,mitras@illinois.edu

Abstract. We describe the design of physical coordination and control language (PCCL) for writing distributed programs that control physical processes such as robots and mobile agents. The key features of this event-based language includes a distributed shared memory abstraction for coordination and a reach-avoid abstraction for controlling the motion. In this paper we present, the executable formal semantics using the $\mathbb{K}$ framework. As a given PCCL program can interact with different physical environments through different sensors and actuators, the executable semantics is parametrized by the dynamics of the program's environment. With illustrative examples we show how the language can help develop portable distributed applications. The K?? enables state space exploration and can help find subtle bugs.

1 Introduction

Distributed robotics and control systems have applications in manufacturing, transportation, and exploration. Managing and programming ...

1.1 Related Work

In this work, we have presented the formal semantics of a language for distributed agent coordination and control. The main focus of this work was on developing a formal model for asynchronous concurrent applications where communication occurs through shared memory. [?] is a language for asynchronous event-driven programming, which allows the programmer to specify the system as a collection of interacting state machines, which communicate with each other using events as opposed to shared memory updates as in PCCL . In an actual implementation (like StarL), agents do respond to message events, which could in principle be modeled in P. Our work provides a framework that allows a relatively novice user to write pseudocode without being concerned with implementation details. There are languages such as Esterel [?] , Lustre [?] and Signal[?]. As in our model of time evolution, these languages also follow a model where time advances in steps. However, since we express our semantics in the $\mathbb{K}$ framework, we can explore various interleaving semantics. In these languages, given a state and an input at the current time step, there is a unique possible state at the next

time step. The synchronous model has the advantage that every event sent to machine is handled in the next clock tick, and is widely used in hardware and embedded systems. However, in an OS or a distributed system, it is impossible to have all the components of the system clocked using a global clock, and hence asynchronous models are used for these systems, which gives a language like P an advantage over PCCL . In such models events are queued, and hence can be delayed arbitrarily before being handled. Theoretically, we can also model delays like this by enforcing application of specific rewrite rules repeatedly, but that would limit the generality of the semantics.

2 Language and system overview

In this section, we give an overview of the system architecture within which PCCL programs execute and then discuss key language features with an example.

We call an entity executing a PCCL program an *agent* or a *process*. The hardware abstraction on which PCCL programs executes includes (a) a controller, (b) shared memory, in addition to the usual (c) local memory and processing unit of the agent. The controller receives lists of way-points and obstacles from the agent's program, drives the actuators (e.g. motors) to reach the way-points while avoiding the obstacles using sensors (e.g. GPS), and updates certain flags to indicate its status to the program. The shared memory abstraction provides single-writer and multi-writer distributed shared variables using which an agent's program can communicate with another agent's program.

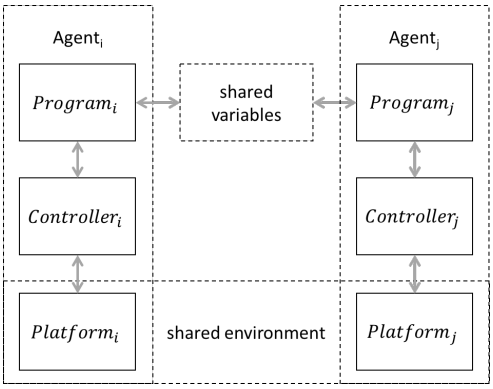


Fig. 1. Architecture of distributed system. Agent programs interact through shared variables. Each agent program also sets waypoints for its own controller which control the physical motion of the agent's platform. The agent platforms inhabit a shared physical environment, and therefore, also interact physically.

For this paper, we assume that all agents execute the same PCCL program; each agent knows the IDs of all participants; and there is a common coordinate

system for the physical space within which the agents are operating. A program is a collection of *variable declarations* and *events*. The language provides two types of shared variables: (a) a *multi-writer* shared variable x is declared as allwrite and allows all agents to do reads and writes on x. (b) a *single-writer* shared variable x is declared as allread block, and it creates an array $x\langle\cdot\rangle$ where the i^{th} component $x\langle i\rangle$ can be read by all but can only be written to by agent i.

A PCCL program is organized as a collection of events. Each event nominally has a precondition (or guard) and an effect. The effect is a sequence of program statements and it may be executed only when the precondition holds. An event is said to be enabled if its precondition holds. If multiple events of a given agent program are enabled, then any one of them is chosen. There is a special event called `Init` that is executed when the program starts.

PCCL provides a special operation doReach for programs to interact with the agent's controller (see Figure 1). A call to doReach instructs the controller to move towards a *target* (position or configuration) while avoiding a set of *obstacles*—both specified in the common coordinate system. Successive calls to doReach may update the sequence of targets and obstacles. The controller communicates with the program using two flags: (a) doReach_done is set if a neighborhood of the target is reached, and (b) doReach_fail is set if the controller determined that it cannot reach the target while avoiding the obstacles. Why different font for the flags? Use the same macros/conventions as in the code fragment. Implementations of controllers for different kinds of agents platforms (e.g., ground rovers and quadcopters) provide different best-effort strategies. In defining the semantics of PCCL the controller is treated as a parameter (see Section ??). Need to sharpen this statement.

2.1 An Illustrative Example

We present a simple illustrative example to demonstrate some of the features of the language, and to aid discussion in future sections. We want to design an application where (a group of) robots try to visit a predetermined sequence of waypoints, with predetermined obstacles. We aim to ensure that a robot choosing the next destination will not pick a waypoint that has already been visited by some robot. The code for this application is provided in Figure 5.2.

`ItemPosition` is a built-in datatype which is used to represent the position of the robot (the physical coordinates (x, y, z)). The robots have a shared list of `ItemPositions` (`dests`) which is initialized using the built-in function `getInput`. We define a boolean variable `Pick` which determines whether the robots are in the stage of picking and moving to the current destination (`currentDest`), or removing the current destination from the shared list of destinations, since it was visited. Functions such as `getInput()`, `getObs()` are provided as uninterpreted functions, which can be defined in an external language as long as they return values with consistent types. [1]

[1] Changing this line

```
Agent::Race

allwrite:
  List<ItemPosition> dests
      = getInput();
allread:

loc:
  ObstacleList obs = getObs();
  boolean Pick = true;
  ItemPosition currentDest;
Init:

PickDest():
  pre(Pick);
  eff:
    if (isEmpty(dests)):
    exit();
    else:
      currentDest = head(dests);
      doReach(currentDest,obs);
      Pick = false;

Remove():
  pre(!Pick);
  eff:
    if(doReach_done):
      atomic:
        if(contains(
        dests,currentDest)):
        remove(dests,currentDest);
      Pick = true;
```

Fig. 2. Race Application

The first stage is `PickDest`, when the next destination in the race is set, the robots try to reach it while avoiding the provided obstacles. Then in the `Remove` stage, each robot *atomically* updates the list of destinations to be visited if it reached the current destination. The `atomic` construct ensures mutual exclusion while updating a shared variable. The function `remove` can only remove an item from a list if it contains said item, the execution gets stuck otherwise. With that in mind, we added a check for whether the `currentDest` is contained in `dests` *within* the atomic block; to ensure that between this check and atomically trying to remove the list element, another robot didn't successfully already remove the same element.

3 Preliminaries

3.1 The StarL robotics framework

We are developing PCCL to interface with the Stabilizing Robotics Language (StarL). StarL is primarily in Java, and it provides language constructs for coordination and control across robots.Two key features of StarL are a distributed shared memory (DSM) primitive for coordination and a reach-avoid primitive for control. DSM allows a program to declare program variables that are shared across multiple robots . This enables programs running on different robots to communicate by writing-to and reading from the shared variable.

All the program threads implementing the application, the message channels, as well as the physical environment of the application (robot chassis, obstacles) are modeled as hybrid automata, and the overall system is described by a giant composition of these automata.

4 Formal Semantics

In this section we describe the formal semantics of key language elements. The system consists of N agents $A_1, \ldots, A_N$. The state of the overall system advances by two types of transitions: (a) *program transitions* correspond to agent program events updating agent variables and possibly setting waypoints for the agent's controllers. Program transitions take zero logical time. (b) *environment transitions* correspond to the physical environment of the agent evolving over an interval of time. During this period, the agent platforms may be moved by their controllers, messages implementing the distributed shared memory abstraction are propagated. Environment transitions are external to PCCL , and therefore, in providing an executable semantics these transitions have to be implemented by external function(s). Thus, a given PCCL program may be executed with different external functions, to obtain different executions.

In the `Race` application of Section 2.1, for example, the state updates brought about by the `PickDest` and `Remove` events are program transitions and take zero logical time. In between these transitions, time may elapse and the corresponding change in the position of the agent is determined by its controller, its physical environment, etc., which are external to the program.

4.1 Overview of $\mathbb{K}$

We give the semantics of PCCL using the $\mathbb{K}$ framework [?]. $\mathbb{K}$ is a rewriting-based executable framework for defining language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as analysis tools such as model checkers and deductive program verifiers "for free". For instance, the state-space exploration capability helps debug PCCL programs. For modelling interactions between agents, an appealing aspect of $\mathbb{K}$ is its inherent support for non-determinism. Rewriting logic [?], makes $\mathbb{K}$ suitable for reasoning about distributed systems that are non-deterministic.

In $\mathbb{K}$, a language syntax is defined using conventional Backus-Naur Form (BNF). The semantics is given as a transition system, specifically, as a set of reduction rules over configurations. A *configuration* is an representation of the program code and state. For PCCL , a configuration represents the code and the program state for all the agent, as well as the state of the physical environment (see Figure 1). Components or members of a configuration are called *cells*. The notation $\langle celltype \rangle_{\mathsf{cellname}}$ represents a cell called *cellname* with a value of type celltype. A special cell, named k, contains a list of computation to be executed. Cells may be nested, that is, contain other cells. A *rewrite rule* describes a one-step transition between configurations. Give a simple rule as a figure and refer to it to explain the following notational conventions. The horizontal line represents a reduction what is a reduction? application of a rewrite rule? (i.e. a transition relation). A cell with no horizontal line means that it is read but not changed by the rule.

4.2 Agent Cells and System Configuration

Each agent has an agent cell which stores the program variables of the agent, its execution context, as well as environment variables that are relevant for the agent such as the agent's position. The agent cell consists of the following cells.

- k: agent's own application code
- id: agent's unique integer identifier
- memory: a map from agent's program variables to addresses
- position: agent's current position in space in common coordinates
- counter: counts the number of times the agent's event block has executed
- lockState: Denotes whether or not the agent holds the requested lock. confused about the type idle

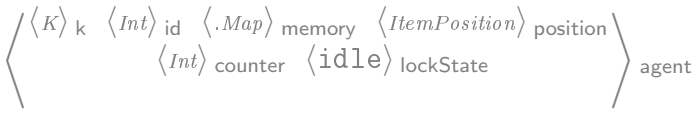


Fig. 3. An agent cell containing other cells.

The memory cell maps *all*—local and shared—variable names to addresses (of type Int). That is, the agent's memory actually has copies of the shared variables. The position cell is specific to agents programs that rely on positional sensors. Other cells for different sensors can be included as well. We will discuss how shared variables reads and writes update the local copies and the related consistency models in Section **??**. We will discuss locking in Section 4.5.

The top-level system configuration cell is called system (Figure 4; it consists of the following main cells:

- agents: contain N agent cells. why *?lets fix N as the number of agents.
- gMemory: maps all shared variables to addresses in the memory
- MMap:a map from memory addresses to variable values
- time: global time elapsed
- counterMap: map of agent ids to their *counter* (Section 4.4) values
- lockQueue: maintains the order of lock requests
- transitionState: indicates whether or not environment transitions are being executed.

4.3 Local Variable Declaration

We first provide the semantic rules for processing variable declarations, as an introduction to reading the semantic rules in $\mathbb{K}$. When a local variable declaration is encountered while an agent's code is being processed, an entry is created in the memory cell, which maps the variable to the next available address in the MMap.

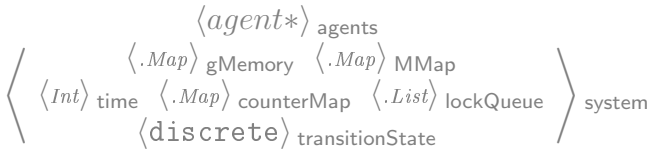
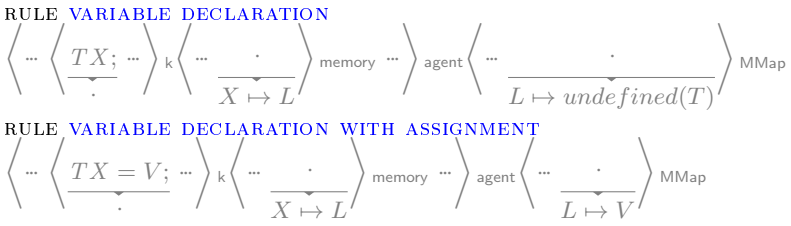


Fig. 4. System configuration.

A corresponding entry is created in the MMap, which maps the aforementioned address to an undefined value of the declaration type if the declaration is uninitialized. It maps to the value the variable was assigned otherwise. We now define the rule for local variable assignment.

Variable declaration rule: The k cell of an agent holds the code being interpreted. Suppose we encounter a local variable declaration T X; where T is the type, and X is the name of the variable. We use the ellipses to represent the fact that (possibly empty) contents of the cell after the element being rewritten do not matter. The variable declaration rule first rewrites the line of code to empty($\cdot$). At the same time, it creates an entry in the memory cell which maps X to L, where L is the next available address in the MMap. Creating an entry in the map is the same as rewriting an empty element to the entry at the end of the map, hence the rewrite rule has the form $\frac{\cdot}{M}$, where M is the entry being created.

We omit the details of how we determine L, as it is not relevant. The rule also creates an entry in MMap with L corresponding to undefined(T) (undefined value of type T). It should be mentioned that we assume unbounded memory



for simplifying the semantics specification.

4.4 Time Advance Semantics

While designing the current semantics of this language we assume that we are able to maintain a global clock (the time cell). The dynamic behavior of each

agent in the system is time varying. Since global time is a part of the system configuration or state, we need to supply rewrite rules for advancing time.

Recall that the event block describes the discrete behavior of each agent. We can say that event block execution takes zero logical time. An event block can execute numerous times between time rewrites. We now define the time model for this semantics. The time model τ is defined as a tuple $\tau = (\delta, n_0)$ where

- δ : is the global time increment.
- n_0 : is the number of times that the event block of each agent must execute before global time is advanced. We use the counter and counterMap cells in the configuration for bookkeeping about this parameter.

We can now provide the rules of time progression in our semantics, given a time model $\tau = (\delta, n_0)$.

Counter Advance Transition: We first discuss how to increment the counter of an agent when it executes an event block once. To ensure that, as soon as the precondition of the event evaluates to true, the rest of the event block should be rewritten to empty. That means, given an event block pre(C); eff : S Es where C is a condition evaluating to true, S is the list of statements in the effect of that event, Es is the list of events following that; this event block rewrites to S. If C evaluates to false, this block rewrites to Es. We skip the details of the rewrite rules involved in this process. endEventBlock is a terminal that we introduce (using a rewrite rule) at the end of the event block. If while executing the semantics, an endEventBlock is encountered in the k cell, it should imply that an event has occurred, the counter should be incremented and the event block should start executing from the top again.

Consider an agent with id i. If an endEventBlock is encountered in its k cell, there is nothing more to do in the current execution. Given further, that the count cell value of the agent is N, the k cell is rewritten to empty. The counter cell is rewritten to $N + 1$, and the corresponding value of its counter in the counterMap cell is also rewritten to $N + 1$. This transition is enabled only as long as N is less than n_0. This ensures that each agent executes its event block exactly n_0 times before the time increments. Note that when the endEventBlock

RULE COUNTER ADVANCE TRANSITION

$$\left\langle \left\langle \underset{\cdot}{\texttt{endEventBlock}} \right\rangle_{\mathsf{k}} \langle i \rangle_{\mathsf{id}} \left\langle \underset{N +_{Int} 1}{N} \right\rangle_{\mathsf{counter}} \right\rangle_{\mathsf{agent}} \left\langle \cdots \underset{i \mapsto N +_{Int} 1}{i \mapsto N} \cdots \right\rangle_{\mathsf{counterMap}}$$

requires $N <_{Int} n_0$

is encountered, we rewrite the entire k cell to empty. How does the next iteration of the event block start then? We maintain a copy of the original application code in the state information, which is copied into the k cell at the beginning of every iteration of the event block. We have not shown this (along with other implementation details) in the interest of clarity of expression.

Time Advance Transition: Suppose that the counterMap cell, which maintains a map of ids to counter values of all agents, is CM. findMax and findMin are functions for computing the maximum and minimum in the range of a map respectively. If both the minimum and maximum values are equal, then all values in the range of the map are equal. The time advance transition says that, given that the current global time is T, if all the counter values in CM are equal to n_0, the global time is rewritten to $T + \delta$. The transitionState is set to `dynamic` from `discrete`. We do not show the rest of the configuration items because they do not appear in the condition or change with the time progression rule. . It

RULE TIME ADVANCE TRANSITION

$$\langle CM \rangle_{\text{counterMap}} \left\langle \frac{T}{T +_{Int} \delta} \right\rangle_{\text{time}} \left\langle \frac{\texttt{discrete}}{\texttt{dynamic}} \right\rangle_{\text{transitionState}}$$

requires $n_0 ==_{Int}$ `findMin` (CM) `and` $n_0 ==_{Int}$ `findMax` (CM)

is worth mentioning that while we designed the time progression such that the event block executes exactly n_0 number of times for each agent, we can easily change the design to accommodate different values for different agents. To do that, we modify the time model by adding another parameter ρ to it. ρ is a map from agent ids to integers. Instead of incrementing the counter by one every time an event occurs, agent with it i increments its counter by $\rho[i]$. The number of times its event block executes is then $\lfloor \frac{n_0}{\rho[i]} \rfloor$.

4.5 Locking

We provide global locks to implement mutual exclusion for the atomic construct. These locks have two major properties:

- At any time, at most one agent can hold a lock.
- A agent needs to request a lock at most once before being eventually granted the lock.

The first property ensures mutual exclusion and the second ensures that high frequency agents do not monopolize the lock. While defining the semantics it is easier for us to present the rules in terms of a single global lock on all multi-writer shared variables. This means that whenever an agent requests a lock it obtains a lock on all shared variables declared in the `MW` block. Each agent has a cell called lockState, which is set to idle if the agent is not requesting or holding the lock, request if the agent has requested the lock but not been granted it yet, and lock if the agent is currently holding the lock. We maintain the lock queue or the order in which requests to the lock were made by agents, in the cell lockQueue.

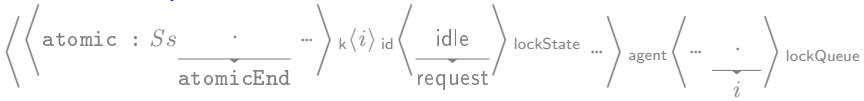
When an agent requests a lock, its id is added at the end of the lockQueue An agent which has requested a lock, is granted it if it is the first in the lock queue. Once an agent has requested a lock, its lockState is updated to request.

An agent is not allowed to add itself to the lockQueue unless its lockState is idle. When the agent is at the beginning of the lockQueue and its lockState is request, then it can execute its atomic block. Once it has done so, it updates its lockState to idle again, and removes itself from the lockQueue.

An agent is in the process of executing the effect of an event whose precondition evaluated to true, when it might need to request a lock. It is however, not allowed to execute the next line of code until it is granted the lock and can proceed. We force the counter to be incremented every time the agent checks its position in the lockQueue. The reason for this is to capture behavior where the lock may not be granted immediately. From our discussion in the previous section, the event block execution takes zero logical time. If the counters were not incremented while the agents were requesting a lock, all locks would be granted before time was incremented from the current value. This would then imply that all locks are granted while zero logical time passed, which is unrealistic behavior. We provide the rewrite rules governing locking below, again, given a time model $\tau = (\delta, n_0)$.

Lock Request Transition: The lock request transition is enabled when an agent with id i, encounters an atomic block containing statements Ss, when its lockstate is idle. Then, the agent adds a terminal atomicEnd just after the atomic block, to ensure that the agent releases the lock after executing it. At the same time, the agent adds itself to the end of the lock queue. The ellipses represent the fact that the lock queue may or may not be empty, it doesn't affect the application of the current rule. Again, we omit showing the irrelevant cells in this rule.

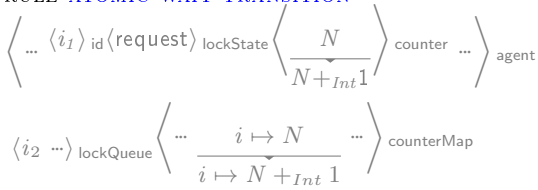
RULE LOCK REQUEST TRANSITION



Atomic Wait Transition: As indicated in above, we capture the fact that locks may not be granted immediately by enabling counter increment transitions when the lockState is request and the agent id is not at the head of the lockQueue. The rewrite rules in the semantics can fire whenever they are enabled, which means that this might result in an agent just continually incrementing its counter, and never doing anything else; thus simulating a failed or crashed agent. Since failure is only observed when communication fails, it can be assumed that the agent failed when it itself fails to communicate.

Atomic Execution Transition: The agent aquires the lock when it is at the beginning of the lockQueue and starts processing the statements inside the atomic block. The counter does not increase in this rewrite, because once the lock has been acquired, the agent can immediately execute the atomic block.

RULE ATOMIC WAIT TRANSITION

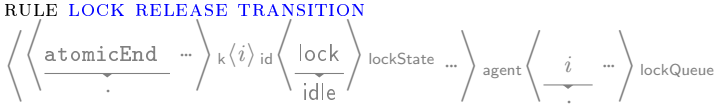


requires $i_1 =_{Int} i_2$ and $N <_{Int} n_0$

RULE ATOMIC EXECUTION TRANSITION

Lock Release Transition Once the atomic block has been executed, the lock must be released, the agent must take itself out of the lockQueue, and set its own lockState to idle. The rest of the event continues to execute. Recall that in the lock request transition, we added a terminal atomicEnd after the atomic block. We can now use that to identify where the atomic block ends.

RULE LOCK RELEASE TRANSITION



4.6 Dynamics

We first present the semantic rules involving the doReach abstraction.

doReach transition The doReach component interfaces with the application using flags doReach_done and doReach_fail. When doReach is invoked, the blackbox containing the implementation of doReach is called. Recall that doReach takes two arguments, the *target*, and the *obstacle*. The exact format of the obstacle is irrelevant to the semantics, as it is used by implementation specific objects in the doReach blackbox. doReach is technically invoked at each time increment. The target and the obstacles are set to current position and empty initially, unless the application specifies otherwise. Again, we omit the details of how we store these in the configuration. We now provide the rule to process the doReach statement when it appears in the application.

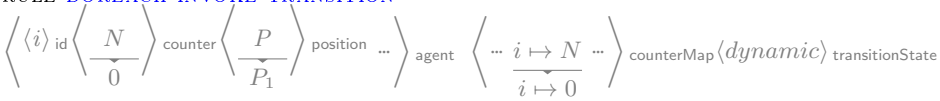
doReach update transition: When a doReach statement is encountered in the program, the statement itself is rewritten to empty. The $\curvearrowright$ is used to break down the program, which is seen as a single task, into a sequence of tasks. Therefore,

it means after rewriting the doReach to empty, the doReach flags should be reset(doReach_done,doReach_fail are both set to false), which should in turn be followed by setting the target of this agent, and the list of obstacles. The setTargetObs function sets the values of target, obstacle , and current time.

RULE DOREACH UPDATE TRANSITION

doReach invoke transition Recall that the transitionState cell of the system is set to dynamic when all the counter values of all agents reach n_0. Suppose the current time is T_0 The time advances by δ. We then invoke the blackbox doReachBB to compute the new contents of the position cell of each agent at time $T_0 + \delta$, given that the position of the agent at time T_0 was P, the target ItemPosition is T, obstacles were stored in O, and the time increment was δ. We omit the details of maintaining T_0, O and T.

RULE DOREACH INVOKE TRANSITION



requires $N ==_{Int} n_0$ `and` $P_1 =$`doReachBB` $(P, T, O, T_0, \delta, i)$

4.7 Syntax

This section describes the formal syntax of PCCL . We first provide the major features of the formal syntax which describe program structure, event structure, and statement structure. As mentioned in the overview, each program consists of three major parts, with variable declarations, an initialization block and an event block. Aside from usual data types and arrays, we provide support for declaring enumerated types, as it is easy to use them as "stages" in applications.[2]

Events, as mentioned earlier are specified by precondition-effect blocks, where the precondition is a boolean expression, and effect blocks contain statements, which can be assignment statements, `if-then-else` statements, `Atomic` statements, function calls, or loops. We omit the productions for the more obvious syntactic elements like expressions, assignment statements, loops, etc.[3]

[2] https://github.com/ritwika314/RoLang/StarL/HLL/Examples/LeaderElect
[3] https://github.com/ritwika314/RoLang/StarL/HLL/Semantics/agent-syntax.k

⟨*Pgm*⟩ :: ⟨*VarDecls*⟩
⟨*EventBlock*⟩

⟨*VarDecls*⟩ :: ⟨*MwDecls*⟩
⟨*LocDecls*⟩

⟨*MwDecls*⟩ :: `MW :` ⟨*Decls*⟩

⟨*SwDecls*⟩ :: `SW :` ⟨*Decls*⟩

⟨*LocDecls*⟩ :: `Loc :` ⟨*Decls*⟩

⟨*Decls*⟩ :: ⟨*Decl*⟩ ⟨*Decls*⟩
| ⟨*Empty*⟩

⟨*Decl*⟩ :: ⟨*EnumDecl*⟩ `;`
| ⟨*ArrayDecl*⟩ `;`
| ⟨*Type*⟩ ⟨*Var*⟩ `;`
| ⟨*Type*⟩ ⟨*Var*⟩ `=` ⟨*Expr*⟩

⟨*InitBlock*⟩ :: `Init :` ⟨*Stmts*⟩

⟨*InitBlock*⟩

⟨*SwDecls*⟩ ⟨*EventBlock*⟩ :: `EventBlock:` ⟨*Events*⟩

⟨*Events*⟩ :: ⟨*Event*⟩ ⟨*Events*⟩ | ⟨*Event*⟩

⟨*Event*⟩ :: ⟨*EventName*⟩ `(` ⟨*Expr*⟩ `) Pre (` ⟨*Expr*⟩`) ; Eff :` ⟨*Stmts*⟩

⟨*Stmts*⟩ :: ⟨*Stmt*⟩ ⟨*Stmts*⟩
| ⟨*Empty*⟩

⟨*Stmt*⟩ :: ⟨*Assignment*⟩
| ⟨*If-Then-Else*⟩
| ⟨*Loop*⟩
| ⟨*Atomic*⟩ | ⟨*FunctionCall*⟩

⟨*Atomic*⟩ :: `Atomic :` Stmts

Fig. 5. Language Syntax Features

5 Experiments

We show two of the case studies we performed using the executable semantics we just defined.

5.1 Fischer's Protocol

Agents try to access a critical section mutually exclusively. Each agent is defined as follows. The agents have a shared variable called reqid which is used to request entry into the critical section. Another shared variable k is used to define wait times and entry times. Each agent initially checks waits for a time between 0 and k, (tracked by c1, and if the reqid is not set (reqid is -1), then it sets reqid atomically to its own id. It then waits for d time (tracked by c2), and checks whether the reqid is its own id. If that is the case, then it enters the critical section, otherwise it goes back to waiting. If a robot enters the critical section, we make it spend cstime(tracked by c3 units of time in the critical section to help detect mutual exclusion violations more easily.

Since the passage of time should correspond to the increments of the variables c1,c2 and c3, we set n_0 to 1, so that the time increments every time the counter of all agents increments.

To ensure that this protocol works, the local inCs variable of at most one agent can be true at time T. The assignment stage takes at least 2 increments of time, as we increment the counter every time an agent makes a lock request, and in this example, the time also increments as soon as the counter is incremented. The following execution trace shows that for d $= 2$, we can violate this property. For

```
Agent::Fischer

allwrite:
  int reqid = -1;
  int k = 10;
  int d = 2;
allread:

loc:
  int id = getAgentIndex();
  int waitTime = rand(k);
  bool start = true;
  bool assign = false;
  bool delay = false;
  bool wait = false;

  bool inCs = false;
  int c1 = 0;
  int c2 = 0;
  int cstime = rand(5);
  int c3 = 0;

Init:

Start():
  pre(start);
  eff:
    if (reqid != -1):
      c1 = 0;
    else:
      start = false;
      wait = true;

Wait():
  pre(wait);
  eff:
      if(c1 < waitTime):
          c1 = c1+1;
      else:
        c1 = 0;
        wait = false;
        assign = true;

Assign():
  pre(assign);
  eff:
    atomic:
     reqid = id;
    delay = true;
    assign = false;

Delay():
  pre(delay);
  eff:
    if (c2 < d):
      c2 = c2+1;
    else:
      if (reqid != id):
        c2 = 0;
        c1 = 0;
        delay = false;
        wait = true;
      else:
        inCs = true;
        wait = false;

InCs():
  pre(inCs);
    eff:
      if (c3 < 3):
          c3 = c3 + 1 ;
        else:
          c3 = 0;
          inCs = false;
          start = true;
          atomic:
            reqid = -1;
```

Fig. 6. Fischer's protocol

bounded executions, we were unable to find traces which violated this property when d was set to a value greater than 2 , which is expected. At time = 0; both the agents have executed the Start event, indicated by the fact that the start variables of both agents, corresponding to addresses 8 and 22 respectively, map to `false` in the MMap. The waitTime for agent with agent 0 is stored at address 7, and is seen from the MMap to be 6. The waitTime for agent 1 is seen to be 3.

At time = 4, agent 1 is seen to be executing Assign event, as evidenced by the fact that variable assign for agent 1 stored at address 23 is true. Agent 0 is still waiting to start its Assign event, since its c1 is still 4 and its waitTime is 6. At time = 11, we see that the inCs values of both agents are true, which violates the correctness of this protocol. During this execution, as agent 0 was assigning the reqid to its own id, agent 1 had entered the delay state. When the agent 1 checked whether it can enter the critical section, agent 0 hadn't assigned reqid to its own it. Agent 1 set inCs to true, which signifies that it is in the critical section, and still is in it when agent 0 enters after satisfying all requirements.

5.2 Race with non-atomic check

We revisit the race example from earlier, where a group of robots try to race to a predetermined set of points. In the PickDest state, the robots choose the next destination to race to.Then in the Remove stage, each robot *atomically* updates the list of destinations to be visited if it reached the current destination. Recall that we put the check for whether an element is present in the shared list dests inside the atomic block. We now perform experiments with two versions of doReachBB, the physical control black box, where the atomic block only contains the update, but not the membership checking.

Figure ?? shows an execution in which the agent cannot process the statement which asks to remove an ItemPosition from the dests, as between the time that it checked for membership and tried to update the list atomically, another agent already managed to remove the said ItemPosition. Figure ?? shows the final configuration of an execution with a different blackbox, which managed not to encounter this error.

```
<agent> ... <memory> start |-> 8 wait |-> 11 inCs |-> 12 cstime |-> 15
c1 |-> 13 c3 |-> 16 reqid |-> 3 id |-> 6 k |-> 4 waitTime |-> 7 assign |-> 9
delay |-> 10 d |-> 5 c2 |-> 14
</memory> <id> 0 </id>
</agent>
<agent> ... <memory> start |-> 22 wait |-> 25 inCs |-> 26 cstime |-> 29
c1 |-> 27 c3 |-> 30 reqid |-> 17 id |-> 20 k |-> 18 waitTime |-> 21
assign |-> 23 delay |-> 24 d |-> 19 c2 |-> 28
</memory> <id> 1 </id>
</agent>
<gMemory> reqid |-> 0 k |-> 1 d |->  2 </gMemory>
<MMap> 0 |-> -1 1 |-> 10 2 |-> 2 3 |-> -1 4 |-> 10 5 |-> 2 6 |-> 0 7 |-> 6
8 |-> true 9 |-> false 10 |-> false 11 |-> false 12 |-> false 13 |-> 0
14 |-> 0  15 |-> 3 16 |-> 0 17 |-> -1 18 |-> 10 19 |-> 2 20 |-> 1
21 |-> 3 22 |-> true 23 |-> false 24 |-> false 25 |-> false 26 |-> false
27 |-> 0 28 |-> 0 29 |-> 5 30 |-> 0
</MMap>
<time> 0 </time>
```

```
<agent> ... <memory> start |-> 8 wait |-> 11 inCs |-> 12 cstime |-> 15
c1 |-> 13 c3 |-> 16 reqid |-> 3 id |-> 6 k |-> 4 waitTime |-> 7 assign |-> 9
delay |-> 10 d |-> 5 c2 |-> 14
</memory> <id> 0 </id>
</agent>
<agent> ... <memory> start |-> 22 wait |-> 25 inCs |-> 26 cstime |-> 29
c1 |-> 27 c3 |-> 30 reqid |-> 17 id |-> 20 k |-> 18 waitTime |-> 21
assign |-> 23 delay |-> 24 d |-> 19 c2 |-> 28
</memory> <id> 1 </id>
</agent>
<gMemory> reqid |-> 0 k |-> 1 d |->  2 </gMemory>
<MMap> 0 |-> -1 1 |-> 10 2 |-> 2 3 |-> -1 4 |-> 10 5 |-> 2 6 |-> 0 7 |-> 6
8 |-> false 9 |-> false 10 |-> false 11 |-> true 12 |-> false 13 |-> 4
14 |-> 0  15 |-> 3 16 |-> 0 17 |-> -1 18 |-> 10 19 |-> 2 20 |-> 1
21 |-> 3 22 |-> false 23 |-> true 24 |-> false 25 |-> false 26 |-> false
27 |-> 0 28 |-> 0 29 |-> 5 30 |-> 0
</MMap>
<time> 4 </time>
```

```
<agent> ... <memory> start |-> 8 wait |-> 11 inCs |-> 12 cstime |-> 15
c1 |-> 13 c3 |-> 16 reqid |-> 3 id |-> 6 k |-> 4 waitTime |-> 7 assign |-> 9
delay |-> 10 d |-> 5 c2 |-> 14
</memory> <id> 0 </id>
</agent>
<agent> ... <memory> start |-> 22 wait |-> 25 inCs |-> 26 cstime |-> 29
c1 |-> 27 c3 |-> 30 reqid |-> 17 id |-> 20 k |-> 18 waitTime |-> 21
assign |-> 23 delay |-> 24 d |-> 19 c2 |-> 28
</memory> <id> 1 </id>
</agent>
<gMemory> reqid |-> 0 k |-> 1 d |->  2 </gMemory>
<MMap> 0 |-> 0 1 |-> 10 2 |-> 2 3 |-> 0 4 |-> 10 5 |-> 2 6 |-> 0 7 |-> 6
8 |-> false 9 |-> false 10 |-> false 11 |-> false 12 |-> true 13 |-> 0
14 |-> 0  15 |-> 3 16 |-> 0 17 |-> 0 18 |-> 10 19 |-> 2 20 |-> 1
21 |-> 3 22 |-> false 23 |-> false 24 |-> false 25 |-> false 26 |-> true
27 |-> 0 28 |-> 0 29 |-> 5 30 |-> 3
</MMap>
<time> 11 </time>
```

```
Agent::Race

allwrite:
  List<ItemPosition> dests
      = getInput();
allread:

loc:
  ObstacleList obs = getObs();
  boolean Pick = true;
  ItemPosition currentDest;
Init:

PickDest():
  pre(Pick);
  eff:
    if (isEmpty(dests)):
    exit();
    else:
      currentDest = head(dests);
      doReach(currentDest,obs);
      Pick = false;

Remove():
  pre(!Pick);
  eff:
    if(doReach_done):
      atomic:
        if(contains(
        dests,currentDest)):
        remove(dests,currentDest);
      Pick = true;
```

Fig. 7. Race Application