1 Preliminaries

1.1 Overview

PCCL allows users to write applications that will run on a distributed system of agents. The user writes a program as though for a single agent, and all agents execute the same code. A PCCL program is a collection of *declarations* and *events*. Shared variables are used for communication between agents. They are declared in an `MW` block. `MW` stands for *multi-writer*, implying that all agents can read from, and write to the variables declared in this block. We provide another type of shared variables called *shared single-writer* variables, which all agents can read from, but only one agent can write to. These variables are parameterized by the `agent index`, an integer which is a unique identifier for each agent in the system, and they are declared in the `SW` block. Local variables are declared in `Loca`l declaration blocks.

As mentioned earlier, PCCL uses a *precondition-effect* style of programming. These precondition and effect statements form events. Each application consists of a special `Init` event, and an `EventBlock`. The `Init` event occurs at the beginning when the application starts executing, and it contains all statements which need to be executed only once; for instance, initializing a shared array. After the `Init` event is executed, the `EventBlock` starts executing. It contains a list of events that define the behavior of the application. The preconditions of each of the events inside the `EventBlock` are evaluated in order of appearance, and the effect is executed if the precondition becomes true. The `EventBlock` can be seen as a (potentially) infinite while-loop.

We also provide an abstraction to manage the physical control of the agents, called `doReach`, which takes as input a *target* to reach, and a list of (predetermined) *obstacles* which need to be avoided. We do not need to specify the format of the obstacles, as different implementations of this `doReach` can have different specifications, but the target in general has the same type as the time varying variables of the system. `doReach` communicates with the program using two flags, *doReach_ done*, and *doReach_fail*. If the target is reached, the *doReach_ done* is set to true, and the *doReach_fail* is set to true when the agent does not seem to have reached the target.

The next section presents the formal syntax, and an example to illustrate the structure of a general application.

1.2 Syntax

1.3 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 as follows.

```
Agent::Race

MW:
    List<ItemPosition> dests = getInput();
SW:

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

Pick():
      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;
```

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]

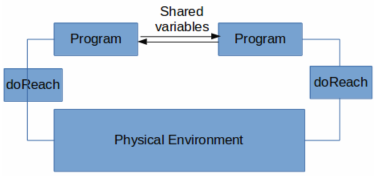
When the next destination in the race is set, the robots try to reach it while avoiding the provided obstacles. Then in the Remove stage, each robot

[1] Changing this line

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.

2 Formal Semantics

In this section we describe the formal semantics of some major language elements. In doing so we use a parametrized formal semantics, where the parameters govern the intervals at which the system is observed(Section 2.3) and how the system evolves with respect to time(Section **??**). Essentially, we specify the semantics of the language, provided there is a procedure to compute the physical control component of a system. For instance, in the race application described in Section 1.3, we parameterize the language semantics with the procedure for computing the positions of the robots at a given time. The event block is like the discrete step of a hybrid automaton, and the dynamic behavior is the continuous trajectory[**?**]. The discrete part of the system corresponds to the programs, which interact with the physical environment using the doReach abstraction. The programs communicate with each other using shared variables.



We explain this further in the following sections, starting with a description of the system state. We then describe the major features of the language semantics, including the time model and the dynamics model, which parametrize the semantics.

A system consists of **N** agents A_1, ..., A_N. In the implementation, we assume that this N is known beforehand. In Section **??**, we described the structure of applications written in this language; each application is written in terms of events. We say that an event has occurred if its precondition was evaluated as true, and the corresponding effect was executed. Once an event has occurred, the program control goes back to the top of the event block. The event block (eventBlock) is said to be executed when any event in the event block occurs once. Given an agent running an application, its event block can be executed repeatedly while the application continues to run. Communication between agents takes place through shared memory updates. Shared memory updates require mutual exclusion, which is implemented by a locking mechanism. We describe the `atomic` construct, which is used for mutual exclusion, in Section 3.

2.1 Configuration

The configuration, or state, holds program variables, objects, and the execution context. We omit implementation minutiae, and only show relevant features. In the configuration, an agent is represented by an agent cell. Recall that a cell is a configuration unit, which is denoted by $\langle\text{cell-type}\rangle_{\text{cell-name}}$. For instance, a cell maintaining the position of an agent, which has type `ItemPosition` will be represented as $\langle\texttt{ItemPosition}\rangle_{\mathsf{position}}$. Cells can also be composite cells, or containers of other cells. They will just be denoted by $\langle\text{cell-contents}\rangle_{\text{cell-name}}$.

Each agent is represented by a composite cell, because the code is executed at agent level, and requires agent-specific information. An agent is identified by a unique identifier id (which is an integer). Its configuration contains the following cells.

- k : Contains an agent's own application code. Each line in this cell is rewritten (possibly to empty) as the semantics is executed.
- id : Contains the agent's unique integer identifier.
- memory : Contains a map from variables to addresses.
- position : The current position of the agent. Note that we only include the position cell in the current semantics because the applications we intend to write are in the domain of robotics. We can create cells to time-varying attributes of the agents as required.
- counter : The counter cell of each agent in the configuration maintains the number of times the event block of that agent has executed.
- lockState : Denotes whether or not the agent holds the requested lock. We discuss locking in Section 3.

$$\left\langle \langle K\rangle_{\mathsf{k}}\ \langle Int\rangle_{\mathsf{id}}\ \langle .Map\rangle_{\mathsf{memory}}\ \langle ItemPosition\rangle_{\mathsf{position}}\ \langle Int\rangle_{\mathsf{counter}}\ \langle \texttt{idle}\rangle_{\mathsf{lockState}} \right\rangle_{\mathsf{system}}$$

Fig. 1. Agent Configuration

Each agent's memory cell is a map from *all* variables(of type Id, short for string identifier), to addresses(of type Int). Both global and local variables are included in this memory. This amounts to maintaining local copies of global variables, which are consistent across agents.

The top level cell is called the system cell, which represents the distributed agent system. This cell contains the following cells:

- agents : This is a container cell for agent cells. It can contain as many such cells as the number of agents running the application, denoted by multiplicity "*".

- gMemory : This is the global memory, contains a map from all shared variables to addresses in the memory .
- MMap : Each variable of the application has an address in memory; we will presently discuss the scope of said memory. MMap stands for memory map, which is a map from addresses in the memory to actual values corresponding to the variables.
- time : This is cell for maintaining time elapsed.
- counterMap : This cell maintains the map of agent ids to their *counter* (Section 2.3) values. We explain what that refers to, in the next paragraph.
- lockQueue: This cell is used to maintain the order in which the lock requests have been made.
- transitionState : This cell stores whether the system is currently demonstrating dynamic behavior or not.

$$\left\langle \begin{array}{c} \langle agent* \rangle_{\mathsf{agents}} \\ \langle .Map \rangle_{\mathsf{gMemory}} \; \langle .Map \rangle_{\mathsf{MMap}} \; \langle Int \rangle_{\mathsf{time}} \; \langle .Map \rangle_{\mathsf{counterMap}} \; \langle .List \rangle_{\mathsf{lockQueue}} \; \langle \mathtt{discrete} \rangle_{\mathsf{transitionState}} \end{array} \right\rangle_{\mathsf{system}}$$

Fig. 2. High Level Semantics Configuration

2.2 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. 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(·). 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 $\underset{M}{\underline{\;\;\cdot\;\;}}$, 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).

RULE VARIABLE DECLARATION

$$\left\langle \cdots \left\langle \frac{TX;}{\cdot} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots \frac{\cdot}{X \mapsto L} \right\rangle_{\mathsf{memory}} \cdots \right\rangle_{\mathsf{agent}} \left\langle \cdots \frac{\cdot}{L \mapsto undefined(T)} \right\rangle_{\mathsf{MMap}}$$

RULE VARIABLE DECLARATION WITH ASSIGNMENT

$$\left\langle \cdots \left\langle \frac{TX = V;}{\cdot} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots \frac{\cdot}{X \mapsto L} \right\rangle_{\mathsf{memory}} \cdots \right\rangle_{\mathsf{agent}} \left\langle \cdots \frac{\cdot}{L \mapsto V} \right\rangle_{\mathsf{MMap}}$$

It should be mentioned that we assume unbounded memory for simplifying the semantics specification.

2.3 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 may or may not be any more code following that, but it cannot change the current event block 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.

RULE COUNTER ADVANCE TRANSITION

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

requires $N <_{Int} n_0$

Note that when the endEventBlock 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 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

RULE TIME ADVANCE TRANSITION

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

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

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 $\lfloor \frac{n_0}{\rho[i]} \rfloor$.

3 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

$$\left\langle \left\langle \mathtt{atomic\ :}\ Ss\ \dfrac{\cdot}{\mathtt{atomicEnd}} \cdots \right\rangle_{\mathsf{k}} \langle i \rangle_{\mathsf{id}} \left\langle \dfrac{\mathsf{idle}}{\mathsf{request}} \right\rangle_{\mathsf{lockState}} \cdots \right\rangle_{\mathsf{agent}} \left\langle \cdots \dfrac{\cdot}{i} \right\rangle_{\mathsf{lockQueue}}$$

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.

RULE ATOMIC WAIT TRANSITION

$$\left\langle \cdots \langle i_1 \rangle_{\mathsf{id}} \langle \mathsf{request} \rangle_{\mathsf{lockState}} \left\langle \dfrac{N}{N +_{Int} 1} \right\rangle_{\mathsf{counter}} \cdots \right\rangle_{\mathsf{agent}} \langle i_2 \cdots \rangle_{\mathsf{lockQueue}} \left\langle \cdots \dfrac{i \mapsto N}{i \mapsto N +_{Int} 1} \cdots \right\rangle_{\mathsf{counterMap}}$$

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

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 EXECUTION TRANSITION

$$\left\langle \left\langle \dfrac{\mathtt{atomic\ :}\ Ss}{Ss} \cdots \right\rangle_{\mathsf{k}} \langle i \rangle_{\mathsf{id}} \left\langle \dfrac{\mathsf{request}}{\mathsf{lock}} \right\rangle_{\mathsf{lockState}} \cdots \right\rangle_{\mathsf{agent}} \langle i \cdots \rangle_{\mathsf{lockQueue}}$$

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

$$\left\langle \left\langle \frac{\texttt{atomicEnd}}{\cdot} \;\cdots \right\rangle_{\mathsf{k}} \langle i \rangle_{\mathsf{id}} \left\langle \frac{\mathsf{lock}}{\mathsf{idle}} \right\rangle_{\mathsf{lockState}} \cdots \right\rangle_{\mathsf{agent}} \left\langle \frac{i}{\cdot} \;\cdots \right\rangle_{\mathsf{lockQueue}}$$