# PGCD: Robot Programming and Verification with Geometry, Concurrency, and Dynamics

Gregor B. Banušić
MPI-SWS
gbbanusic@gmail.com

Rupak Majumdar
MPI-SWS
rupak@mpi-sws.org

Marcus Pirron
MPI-SWS
mpirron@mpi-sws.org

Anne-Kathrin Schmuck
MPI-SWS
akschmuck@mpi-sws.org

Damien Zufferey
MPI-SWS
zufferey@mpi-sws.org

## ABSTRACT

Robotics applications are typically programmed in low-level imperative programming languages, leaving the programmer to deal with dynamic controllers affecting the physical state, geometric constraints on components, and concurrency and synchronization. The combination of these features —dynamics, geometry, and concurrency— makes developing robotic applications difficult. We present PGCD, a programming model for robotics applications consisting of assemblies of robotic components, together with its runtime and a verifier. PGCD combines message-passing concurrent processes with *motion primitives*, which represent continuous evolution of trajectories in geometric space under the action of dynamic controllers, and explicit modeling of *geometric frame shifts*, which allow relative coordinate transformations between components evolving in space. We describe a verification algorithm for PGCD programs based on model checking and SMT solvers that statically verifies concurrency-related properties such as absence of deadlocks and geometric invariants such as absence of collision during motion. We have implemented a runtime for PGCD programs that compiles down to imperative code on top of ROS and runs directly on robotic hardware. We illustrate the programming model and reasoning principles by building a number of statically verified robotic manipulation programs on top of 3D-printed robotic arm and cart assemblies.

## CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering →** *Formal software verification*; System modeling languages;

## KEYWORDS

cyber-physical systems, domain specific language, composition, communication, geometry, verification

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.
*ICCPS '19, April 16–18, 2019, Montreal, QC, Canada*
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6285-6/19/04…$15.00
https://doi.org/10.1145/3302509.3311052

**ACM Reference Format:**
Gregor B. Banušić, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. 2019. PGCD: Robot Programming and Verification with Geometry, Concurrency, and Dynamics. In *10th ACM/IEEE International Conference on Cyber-Physical Systems (with CPS-IoT Week 2019) (ICCPS '19), April 16–18, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3302509.3311052

## 1 INTRODUCTION

Modern robotics software consists of concurrent communicating processes, each of which executes dynamic controllers (called motion or action primitives). Motion primitives sense and actuate continuous physical processes, which may be coupled, so the execution of one primitive affects the physical state of a concurrently executing primitive. Further, since the robotic components reside in 3D space, the software must ensure that the range of motions executed concurrently by the motion primitives are compatible with the geometry of the components and there are no collisions.

As a example, consider a robotic assembly consisting of a mobile cart with an arm attached to it, as depicted in Figure 1. We illustrate the difficulty of the robotics programmer on this example, for a task *Fetch* which requires the system to fetch an object. We assume both the cart and the arm comes with a set of capabilities: the cart can move between locations and the arm can grab objects in its vicinity. The capabilities are provided as *motion primitives*: abstractions of underlying dynamic controllers. For example, the motion primitive "move" could be implemented as a controller that plans a path and executes the plan by sensing the state and actuating the motor, and "grab" may be implemented as a complex controller that plans a trajectory to grab and lift an object. However, implementing and verifying a task as simple as *Fetch* is complicated for the following reasons.

First, the controllers of the components are dynamically coupled; for example, the movement of the cart depends on the weight and center of mass of the arm: a small, light arm may allow a fast motion planner compared to a heavy or dangly arm. Second, both the arm and the cart live in 3D space, and this influences their range of actions. For example, whether the cart can navigate through a passage may depend on the state of the arm: an extended arm can collide with an obstacle and invalidate a path that the cart, by itself, could traverse. Conversely, the range of motion of the arm is restricted by the base of the cart it is attached to. Third, the motion primitives of the cart and the arm refer to coordinates in their local frame: when the cart moves, the arm moves along and

any communication of the geometric space between the cart and the arm must transform the information between the coordinate systems. Fourth, a natural approach to decompose *Fetch* is for the cart to move close to the target while the arm remains folded, then the arm to grab the object, and finally the cart to move back. This requires synchronization between the code running on the cart and on the arm, to signal which step is currently executed and what each process guarantees the other, and inherits all the complexities of concurrent programming.

The robotics application developer today is left to navigate these complexities by themselves. Programs are developed in imperative languages such as C++ or Python or low-level robotics languages provided by robot manufacturers, such as Rapid from ABB or KRL from Kuka. While there is support for messaging (e.g., the robot operating system (ROS) [23]), there are no existing programming models or tools to simultaneously reason about the interaction between concurrency, geometry, and dynamics.

**Contributions.** In this paper, we develop PGCD, a concurrent programming model which combines Geometric constraint reasoning of the robotic components with message passing Concurrency and motion primitives for Dynamics. A program in our model consists of a set of *processes* —each process represents a logical portion of a robotic assembly ("cart" or "arm"). Processes run sequential code and send or receive messages as well as execute a set of *motion primitives* on the underlying physical state. A program is structurally determined by assembling processes through *attachments*: an attachment couples the physical state of two components and also determines the relative coordinate transformations between their geometries. For instance, for the *Fetch* example, there is a process for the cart and a process for the arm. The program is obtained by attaching the arm process to the cart process. The semantics of PGCD programs define a transition system in which communication occurs in logical "zero time" and motion primitives execute in real time. The semantics include geometric transformations between processes. Thus, the content of messages between the cart and the arm processes is transformed to the recipient's coordinate system.

Second, we develop a verifier for PGCD programs. The verifier takes as input a collection of robotic components, a PGCD program, a specification of the motion primitives, and a description of the environment. The specification of a motion primitive contains both constraints over a robot's state and its *footprint*, i.e., the region of space used by the robot when executing that motion primitive. The verification process has two steps. First, we check correctness of communication and synchronization, such as the absence of deadlock. Second, we ensure concurrent executability of motion primitives through assume-guarantee reasoning and separation of geometric resources. The separation of geometric resources checks the important invariant that different components do not collide: they always reside in disjoint parts of 3D space. The programmer writes constraints and footprints of each process in its local frame; the run time and the verification engine automatically performs frame shifts.

We have implemented our language and analysis in Python and provide a runtime system to execute our programs on top of ROS. We have used our implementation to verify implementations of actual multi-robot co-ordinations. Further, the verified programs execute on real robotic hardware (see Figure 1). Our evaluation shows that our framework for programming of multi-robot co-ordination and manipulation can lead to statically verified implementations that run on off-the-shelf and custom-built robotics hardware platforms.

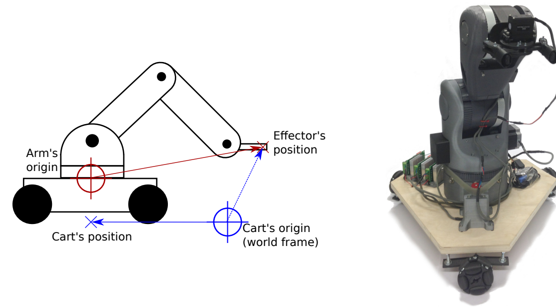



**Figure 1: (a) Schematic and (b) actual cart and arm**

Together, our programming model, runtime system, and verifier lay the foundation for correct design and static verification of complex robotic applications interacting dynamically in geometric space.

**Related Work.** Many computational approaches have been developed for concurrent and real-time communication and computation, but few cover the combination of communication, complex geometry, and dynamic control of physical state. Modeling paradigms for hybrid systems such as hybrid automata and its extensions [1, 2, 20] allow expressive dynamics, but little support for compositional programming and reasoning about communication. Timed extensions to process algebras, Petri nets, or other concurrency models allow the mixing of message passing and time, but do not combine geometric reasoning and resource accounting. For the most part, analysis algorithms for these models are intractable. In principle, logics such as differential dynamic logic [22] and hybrid process algebras [3, 5, 16, 25] enable reasoning about arbitrarily complex concurrent and hybrid programs. However, the primary goal of these systems is interactive verification of models.

Cardelli and Gardner [6] define a process algebra for geometric interaction. Their formalism combines communication and frame shifting, however, they do not consider dynamic flows of geometric objects over time, which is crucial in a robotics context. Moreover, the objective in their process algebra is an abstraction theorem, and not reasoning about programs. Spatial logics have also been explored from a topological perspective [7] where the modal operators describe neighborhood relations. Such a framework can express and check properties about arrangements, but cannot deal with temporal evolutions.

From the perspective of DSLs for distributed robotic systems, recent projects like StarL [18], Drona [8], and Koord [12] integrate a DSL, specification, and verification support in the same framework. StarL programs are composed of coordination and motion primitives which have been specified in an interactive theorem prover which can be used to verify the programs. Drona is built on top of a state-machine based programming language, integrates

a motion planner, and uses a model-checker to test the programs. Koord is event based where events trigger global actions which perform computations and call motion primitives for different robots. The verification uses a bounded model checker or user provided inductive invariants. None of these systems integrate programming and reasoning with concurrency, dynamics, and geometry.

# 2 PRELIMINARIES

## 2.1 Space and Frames

We work with robotic components embedded in 3D Euclidean vector space $\mathbb{R}^3$. Positions are points in $\mathbb{R}^3$, displacements are vectors in $\mathbb{R}^3$, addition + is component-wise, scalar multiplication $\cdot$ with a scalar from $\mathbb{R}$ is component-wise, and norm is defined as usual. While points and vectors are both syntactically represented as three real values, for technical reasons, we distinguish the *sorts* points and vectors. While we assume the distinction is clear from the context, when necessary, we write $V(\mathbb{R}^3)$ for the set of vectors in $\mathbb{R}^3$, which is an isomorphic copy of $\mathbb{R}^3$. We follow the convention of calling $x$, $y$, and $z$ the 1st, 2nd, and 3rd components of a vector.

A *frame* $\mathcal{A}$ is an orthonormal basis in $V(\mathbb{R}^3)$. We call $\mathbf{u}_x$, $\mathbf{u}_y$, and $\mathbf{u}_z$ the 1st, 2nd, and 3rd unit vectors defining a frame.

We consider transformations between frames which are *rigid motions*, i.e., transformations preserving distances, angles, and orientation. The set of rigid motions forms the special Euclidean group $SE(3)$ and the set of 3D rotations gives the subgroup $SO(3)$. A rigid motion $M \in SE(3)$ can be decomposed in a rotation $R \in SO(3)$ and a translation $r \in V(\mathbb{R}^3)$: applying $M$ to a point $p \in \mathbb{R}^3$ gives the point $Rp + r$; applying $M$ to a vector $v \in V(\mathbb{R}^3)$ gives the vector $Rv$, since vectors are invariant under translation.

It is well known that elements of $SO(3)$ can be represented by $3 \times 3$ orthogonal matrices, elements of $SE(3)$ can be represented by $4 \times 4$ matrices [17], and the group operation is matrix multiplication. Thus we identify $SE(3)$ with the set

$$\left\{ \begin{bmatrix} R & r \\ 0_{1\times 3} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4} \middle| R \in \mathbb{R}^{3\times 3} \wedge r \in \mathbb{R}^3 \wedge R^T R = R R^T = I \wedge |R| = 1 \right\}.$$

We write $I$ for the $4 \times 4$ identity matrix. To use this representation, we extend a point $p = (x, y, z)^T \in \mathbb{R}^3$ to $(x, y, z, 1)^T$ and a vector $v = (a, b, c)^T \in V(\mathbb{R}^3)$ to $(a, b, c, 0)^T$.

In what follows, we fix a *world frame* $\mathcal{W}$. Intuitively, the world frame is a coordinate system that remains fixed and to which every other coordinate system can be related.

## 2.2 Motion Primitives

Let $X$ and $W$ be two sets of real-valued variables, representing internal state and external input variables of a dynamical system, respectively. We write $[\![X]\!]$ and $[\![W]\!]$ for the set of (real-valued) valuations to all the variables in $X$ and $W$.

When applying a controller to a dynamical system, the values of the variables in $X$ are updated over time, while respecting the values of variables in $W$, set by the external world. This dynamic process results in a pair of state and input trajectories $(\xi, \nu)$, i.e., a valuation over time to variables in $X$ and $W$ s.t. $\xi$ fulfills a desired property.

We capture the effect of executing a controller on the dynamical system for $T$ time steps by a *motion primitive* $\mathbf{m}$ : (Pre, Inv, Post) which consists of a *pre-condition* Pre $\subseteq [\![X]\!] \times [\![W]\!]$ which gives conditions under which the motion primitive can be applied, an *invariant* Inv $\subseteq ([0,T] \to [\![X]\!]) \times ([0,T] \to [\![W]\!])$ which gives the invariants that hold for the resulting motion, and a *post-condition* Post $\subseteq [\![X]\!] \times [\![W]\!]$ which gives the possible states at the end of the motion. A valid *trajectory* of duration $T$ of a motion primitive $\mathbf{m}$ is a pair of (measurable essentially bounded) functions $(\xi, \nu)$ mapping the real interval $[0,T]$ to $[\![X]\!]$ and $[\![W]\!]$, respectively, such that $(\xi, \nu) \in$ Inv, $(\xi(0), \nu(0)) \in$ Pre, and $(\xi(T), \nu(T)) \in$ Post. We shall represent Pre and Post syntactically as a first-order formula over $X \cup W$, and Inv as a first-order formula over $X \cup W \cup \{t\}$, where $t$ denotes the time variable.

*Example 2.1.* Consider a cart moving on a 2D plane. The physical state of the cart is its geometric center $\mathbf{p}_{\text{cart}} \in \mathbb{R}^3$ and orientation $\mathbf{r}_{\text{cart}} \in SO(3)$ in the world frame and its speed $s_{\text{cart}} \in \mathbb{R}$. It has an external variable $m_{\mathbf{obj}}$, which denotes the mass of the object it is carrying. A trivial motion primitive $\mathbf{idle}(\mathbf{p}_0, \mathbf{r}_0)$ keeps the cart at its current position $\mathbf{p}_0$ and orientation $\mathbf{r}_0$; the pre-condition is $s_{\text{cart}} = 0$ (i.e., it is at rest), the post-condition is $s_{\text{cart}} = 0 \wedge \mathbf{p}_{\text{cart}} = \mathbf{p}_0 \wedge \mathbf{r}_{\text{cart}} = \mathbf{r}_0$, and the invariant is $\mathbf{p}_{\text{cart}}(t) = \mathbf{p}_0 \wedge \mathbf{r}_{\text{cart}}(t) = \mathbf{r}_0$ for all $t \in [0,T]$. A slightly more interesting motion primitive is $\mathbf{forward}(\mathbf{p}_0, \mathbf{r}_0)$, which moves the cart in the direction of its orientation for one unit. The pre-condition is $s_{\text{cart}} = 0 \wedge \mathbf{p}_{\text{cart}} = \mathbf{p}_0 \wedge \mathbf{r}_{\text{cart}} = \mathbf{r}_0 \wedge m_{\mathbf{obj}} \leq m_{max}$, giving an upper bound on the carried mass for motion to be possible. The post-condition is $s_{\text{cart}} = 0 \wedge \mathbf{p}_{\text{cart}} = \mathbf{p}_0 + \mathbf{r}_{\text{cart}} \mathbf{u}_x \wedge \mathbf{r}_{\text{cart}} = \mathbf{r}_0$. The invariant can specify a bound on the velocity, e.g., $0 \leq s_{\text{cart}} \leq v_{\max}$. The motion primitive abstracts from the underlying dynamics of the cart (which would depend on $m_{\mathbf{obj}}$ and the dynamic controller which determines the force applied to the motors to move the cart). ◻

# 3 PGCD PROGRAMS

We introduce PGCD, a concurrent programming model for cyber-physical components embedded in 3-dimensional geometric space. A PGCD program is a set of concurrently executing processes. Each process "controls" a set of physical variables by executing motion primitives on them, but additionally communicates and synchronizes with other processes. The dynamics of the physical variables are coupled. The program structure reflects this coupling, i.e., the composition of processes reflects the couplings and frame shifts between physical variables of their underlying dynamics. Thus, concurrent processes can execute in different reference frames relative to each other; e.g., an arm attached to a cart describes its motion in its local frame, but its local frame can move relative to the world frame if the cart moves.

## 3.1 Syntax

*Processes.* We consider a fixed finite set $C$ of *processes*. Each *process* $P \in C$ is a tuple $(Var, \mathcal{M}, S, \rho, rsrc)$ where $Var$ is a set of variables, with two distinguished disjoint subsets $X$ and $W$ of *physical state* and *external input* variables, $\mathcal{M}$ is a set of motion primitives, $\rho : Var \to [\![Var]\!]$ is a *store* mapping variables to values, $rsrc : \rho \to 2^{\mathbb{R}^3}$ is a *resource function*, and $S$ is a *statement* generated by the grammar:

$$\begin{aligned} S \ ::= \ & x := expr \mid \mathbf{m} \mid \mathbf{send}(a, l, expr) \mid \mathbf{receive}(\mathbf{m})\{(l, x, S)^+\} \\ & \mid S; S \mid \mathbf{if}\ expr\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{skip} \mid \mathbf{while}\ expr\ \mathbf{do}\ S \end{aligned}$$

where $a \in C$ is a (different) component, $\mathbf{m} \in \mathcal{M}$ is a motion primitive, $x \in Var$ ranges over variables, $l$ is a label from a fixed finite set of labels, and *expr* comes from an (unspecified) effectively computable language of arithmetic expressions. For the sake of simplicity, we omit types for variables and assume well-typed processes and motion primitives.

A process represents a unit of a program which controls a continuous physical system through the application of motion primitives and, additionally, communicates with other concurrently executing processes. The *store* $\rho$ gives values to the local process variables *Var*; this includes valuations to the physical state variables. The resource function gives an upper bound on the geometric space used by a process. Part of the statements form a core imperative programming language, with skip, assignments, sequential composition, conditionals, and loops. In addition, a process can execute motion primitives and send and receive *labeled* messages for synchronization. The message labels come from a finite set known to all processes. As shorthand, we often write labeled messages as $l(v)$ where $l$ is the label and $v$ a value. When the message does not carry a value, we simply write $l$. Receiving messages operates on a list of triples of the form $(l, x, S)$ where $l$ is a label, $x$ is a name to which the received value is assigned, and $S$ is the continuation. Since receive is a blocking operation it also takes as argument a motion primitive which is executed while the process waits for a message. We consider messages with a single payload value, but this can be generalized to tuples.

The resource function *rsrc* takes as input the state of the process and returns an over-approximation of the space used by the robot (a subset of $\mathbb{R}^3$).

*Example 3.1.* Consider the cart from Example 2.1. A process corresponding to the cart would "wrap" the motion primitives and additionally implement a control program which determines when to apply the motion primitives. The set $Var = \{\mathbf{p}_{\text{cart}}, \mathbf{r}_{\text{cart}}, s_{\text{cart}}\} \cup \{m_{\mathbf{obj}}\}$ gives the physical state and external input variables, respectively. The program

$$\mathbf{receive}(\mathbf{idle}(\rho(\mathbf{p}_{\text{cart}}), \rho(\mathbf{r}_{\text{cart}})))\{(step, _, \mathbf{forward}(\rho(\mathbf{p}_{\text{cart}}), \rho(\mathbf{r}_{\text{cart}})))\}$$

specifies that the cart remains idle at its current position $\rho(\mathbf{x})$ until it receives a *step* message (without other values), and then steps one unit using the motion primitive **forward**.

Let $d_l, d_w, d_h$ be bounds on the cart's length, width, and height, s.t. their center point coincides with the cart's center. The resource function *rsrc* gives a bounding box around the cart's position:

$$\left\{ p' \in \mathbb{R}^3 \middle| \begin{array}{l} -d_l/2 \le (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_x) \le d_l/2 \\ \wedge -d_w/2 \le (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_y) \le d_w/2 \\ \wedge 0 \le (p' - \mathbf{p}_{\text{cart}}) \cdot (\mathbf{r}_{\text{cart}}\mathbf{u}_z) \le d_h \end{array} \right\} \quad (1)$$

We write the resource in this style, rather than the more obvious predicate $\{x \mid |x_1 - p_{\text{cart}}.x| \le \frac{d_l}{2}, |x_2 - p_{\text{cart}}.y| \le \frac{d_w}{2}, |x_3 - p_{\text{cart}}.z| \le \frac{d_h}{2}\}$ to make frame reasoning about resources easier; in (1), vectors are defined as abstract elements and we can directly transform them across frame shifts. □

*Attached Composition.* Consider two processes $P_1$ and $P_2$ with disjoint sets of variables. The simplest way to compose $P_1$ and $P_2$ is to "connect" some physical variables of one with external variables of the other and vice versa. This couples the motion primitives of the two processes.

A *connection* $\theta$ between $P_1$ and $P_2$ is a finite set of pairs of variables, $\theta = \{(x_i, w_i) \mid i = 1, \ldots, m\}$, such that: (1) for each $(x, w) \in \theta$, we have $x \in P_1.X$ and $w \in P_2.W$ or $x \in P_2.X$ and $w \in P_1.W$, and (2) there does not exist $(x, w), (x', w) \in \theta$ such that $x$ and $x'$ are distinct. Two connections $\theta_1$ and $\theta_2$ are *compatible* if $\theta_1 \cup \theta_2$ is a connection. Given a connection $\theta$, we write $\theta(x) = \{w \mid (x, w) \in \theta\}$ and $rng(\theta) = \{w \mid \exists x.(x, w) \in \theta\}$. Note that in a connection, a physical state variable of a process may be connected to several external variables, but each external variable of a process is connected to at most one state variable.

A connection couples the variables of two processes, but they may be interpreted in different frames. Thus, a connection between the components may require a frame shift when communicating geometric objects. This is the motivation for attached compositions, defined next.

Let $\theta$ be a connection between $P_1$ and $P_2$ and let $M$ be a term over the variables of $P_1$. We assume $M$ evaluates to a frame transformer in $SE(3)$. We define the *attached composition* operation $P_1 \lhd_{\theta,M} P_2$ which connects variables through a connection $\theta$ and applies a frame shift $M$ for any communication of geometric objects (points or vectors) from $P_2$ to $P_1$ and a reverse shift $P_1.\rho(M^{-1})$ for any communication from $P_1$ to $P_2$. The semantic rules in the next section will apply this frame shift automatically.

A connection introduces the following constraint on stores: $P_1.\rho(w) = P_1.\rho(M)(P_2.\rho(y))$ whenever $(y, w) \in \theta$ with $y \in P_2.X$ and $w \in P_1.W$ and $P_2.\rho(w) = P_1.\rho(M^{-1})(P_1.\rho(y))$ whenever $(y, w) \in \theta$ with $y \in P_1.X$ and $w \in P_2.W$. That is, the variables in $P_1.W$ connected to those in $P_2.X$ are "set" by the corresponding values in $P_2$'s store after the frame shift, and vice versa.

Let $P_1$, $P_2$, and $P_3$ be processes. Let $\theta_{12}$ be a connection between $P_1$ and $P_2$ and $\theta_{13}$ a connection between $P_1$ and $P_3$. Let $M_{12}$ and $M_{13}$ be relative frame shifts between $P_1$ and $P_2$ and $P_1$ and $P_3$, respectively. The operation $P_1 \lhd_{\theta,M} P_2$ is considered left associative; we write $P_1 \lhd_{\theta_{12},M_{12}} P_2 \lhd_{\theta_{13},M_{13}} P_3$ for $(P_1 \lhd_{\theta_{12},M_{12}} P_2) \lhd_{\theta_{13},M_{13}} P_3$. In this expression, both $P_2$ and $P_3$ are children of $P_1$. Therefore, we have

$$(P_1 \lhd_{\theta_{12},M_{12}} P_2) \lhd_{\theta_{13},M_{13}} P_3 = (P_1 \lhd_{\theta_{13},M_{13}} P_3) \lhd_{\theta_{12},M_{12}} P_2.$$

However, contrary to the "usual" parallel composition of processes, attached composition is not commutative. The frame of $P_1 \lhd_{\theta,M} P_2$ is the frame of $P_1$. Swapping $P_1$ and $P_2$ would change that unless $M = I$. Hence, we only have a restricted form of commutativity, i.e.

$$P_1 \lhd_{\theta,I} P_2 = P_2 \lhd_{\theta,I} P_1.$$

*Example 3.2.* We consider a second process, an arm which is mounted on the cart. The cart's variables are the three angles $\alpha, \beta, \gamma$ of the joints between each of its parts and $m_{\text{arm}}$ which represent the overall mass of the arm (motion platform, gripper, carried object). The frame of the arm is the center of its base as shown in Figure 1(a). We model this using the attached composition operation $C \lhd_{\theta,M} A$ with $M = \left[\begin{smallmatrix} \mathbf{r}_{\text{cart}} & \mathbf{p}_{\text{cart}} \\ 0_{1\times 3} & 1 \end{smallmatrix}\right]$ which shifts the frame according to the cart's position and heading. $\theta = \{(m_{\text{arm}}, m_{\mathbf{obj}})\}$ connects the arm's mass to the cart's payload. □

*Programs.* A (concurrent) *program* $\Pi$ connects processes using the attached composition operator:

$$\Pi ::= P \mid \Pi \lhd_{\theta,M} \Pi \quad (2)$$

**Algorithm 1:** Cart

```
1  send(arm, fold);
2  receive (idle)
3  | folded ⇒ skip
4  while (|target − p| > reach) do
5  | moveToward(target)
6  send(arm, grab(target));
7  receive (idle)
8  | grabbed ⇒ skip
9  send(arm, fold);
10 receive (idle)
11 | folded ⇒ skip
12 while (p ∉ homeRegion) do
13 | moveToward(homeRegion);
14 send(arm, done);
```

**Algorithm 2:** Arm

```
1  while true do
2  | receive (idle)
3  | | fold ⇒
4  | | | move(origin);
5  | | | send(cart, folded)
6  | | grab(loc) ⇒
7  | | | grab(loc);
8  | | | send(cart,
   | | |   grabbed)
9  | | done ⇒
10 | | | break
```

**Figure 2: Pseudocode for cart and arm**

where $\theta$ is a connection, and $M$ is a term representing a frame transformer. Since attached composition is left associative, a program arranges processes in a tree structure which induces a parent-child relationship between processes.

For simplicity of notation, we assume there is a *virtual process* $\mathcal{V}$ defined as $(_, \emptyset, C_V, \{idle\}, while(true)idle)$ with $C_V = (\emptyset, \emptyset, \emptyset, _ \to \emptyset, _ \to \emptyset)$. This allows us to represent two components $P$ and $P'$ which are not "physically attached" together by attaching both of them to $\mathcal{V}$, i.e., $V \lhd_{\emptyset,M} P \lhd_{\emptyset,M'} P'$. $\mathcal{V}$ may also be used to model features of the environment, e.g., attaching obstacles to it.

*Example 3.3.* Mounting an arm on the cart enables completion of more complex tasks that neither component could achieve alone. We now want to write a concurrent program that controls the arm and the cart such that an object at a remote location is fetched. Algorithms 1 and 2 show two pieces of code to highlight the communication between them. We omit additional details of the computation for simplicity. For readability, we use some syntactic sugar; these programs can be easily compiled into our core grammar for statements. □

## 3.2 Semantics

Intuitively, the execution of PGCD programs happen in rounds. Each round has two sub-rounds. In the first subround, components exchange messages in logical zero time. In the second, each component executes a motion primitive for a time period of duration $T$. The real time execution of motion primitives synchronizes all components.

Formally, we define the semantics of our programming model as labeled transition rules between program states. A program state consists of a program $\Pi$ and the stores of each process in $\Pi$. Given a store $\rho$ of a process and a variable $x \in Var$ of that process, we write $\rho(x)$ for the value of variable $x$ and lift this to expressions: $\rho(e)$ is the evaluation of $e$ in environment $\rho$. We write $\rho(x) \leftarrow v$ for the store which maps variable $x$ to $v$ but agrees with $\rho$ on all other variables.

The transitions $\rightarrow$ are of the form:

- $\xrightarrow{\tau}$: an internal step to a process.
- $\xrightarrow{p!l(v)}$: sending label $l$ with value $v$ to component $p$.
- $\xrightarrow{p?l(v)}$: $p$ receiving a message with label $l$ and value $v$.
- $\xrightarrow{\xi,\nu,T}$: the system follows the output trajectory $\xi$, with external inputs $\nu$ for the duration $T$.

*Contexts.* We define the semantics in a contextual style. A *statement context* $\Sigma$ extracts the next statement to be executed in a sequence "$\Sigma ::= [] \mid \Sigma; S$" and $\Sigma[S]$ is obtained by replacing [] by a statement $S$ in $\Sigma$.

The semantic rules given below use some auxiliary functions and predicates. The *leftH* functions returns the output of the leftmost process in a program. It is recursively computed with (i) $leftH(P) = P.\rho$, (ii) $leftH(\Pi \lhd_{\theta,M} \Pi') = leftH(\Pi)$. We use *leftH* for the frame shift in a composition which is a function of the output of the leftmost process. For example, we write $leftH(\Pi_1 \lhd_{\theta,M} \Pi_2)(M)$ for the transformation in $SE(3)$ obtained by evaluating the term $M$ in the store of the leftmost process in the program. $disjoint(P, Q)$, with $P \lhd_{\theta,M} Q$, is a shorthand for $P.rsrc(P.\rho) \cap leftH(P)(M)(Q.rsrc(Q.\rho)) = \emptyset$. This predicate checks that the footprint of the left and right components is disjoint. Anytime the state of a process can change, disjointness needs to be preserved.

*Transitions.* The main transition rules are presented in Figures 3 and 4. Figure 3 shows transitions relating to inter-process communication. Inter-process communication happens by rendezvous on a shared channel: a sender process sends a value $v$ on a channel $a$ and simultaneously a receiver process receives the value, shifted to its own frame, and continues executing. The semantics take care of the frame shift of value $v$ from the sender to the receiver. This approach of implicitly taking care of the frame shift is used in existing systems like the TF2 library [10].

The (CommSend) and (CommRecv) rules describe how processes send and receive messages. When sending a message, the expression $e$ is evaluated to a value $v$ in the store of $P$ ($\rho(e) \Downarrow v$). Sending does not change the local state, it only consumes the send instruction. Receiving a message binds the value $v$ carried by the message to the receiving variable $x$ in the store ($\rho(x) \leftarrow v$) and continues with the appropriate statement $S$ determined by the label.

The rules (CompL) and (CompR) "propagate" communication and silent steps: if a component of a propagate can make a transition labeled $a$, then the entire program can also make a transition labeled $a$. When propagating the transition, the store is updated to reflect the frame and connections. If the transition carries a message from the right side of the composition, the value in the message is shifted to match the overall frame inherited from the left process ($leftH(P)(M)(v) \Downarrow v'$). Furthermore, the transition may update the local state which changes the resources used by the processes. So we also check that the two components are disjoint.

The (CommSyncLR) and (CommSyncRL) rules match the send and receive statements. The structure is similar to the (CompL) and (CompR) rules with both sides changing. Once the send and receive are matched the label of the action is not propagated further and the action becomes an internal action ($\tau$) of the composed processes.

Figure 4 shows how the motion primitives execute. The transitions are labeled with the trajectory ($\xi$) of the local states, the external inputs ($\nu$), and the total time of the transition ($T$). The

(CommSend)

$$\frac{\rho(e) \Downarrow v}{(Var, \mathcal{M}, \Sigma[\mathbf{send}(a, l, e)], \rho, \cdot) \xrightarrow{a!l(v)} (Var, \mathcal{M}, \Sigma[\mathbf{skip}], \rho, \cdot)}$$

(CommRecv)

$$\frac{\rho' = \rho(x) \leftarrow v}{(Var, \mathcal{M}, \Sigma[\mathbf{receive}(\mathbf{m})\{\ldots(l, x, S)\ldots\}], \rho, \cdot) \xrightarrow{P?l(v)} (Var, \mathcal{M}, \Sigma[S], \rho', \cdot)}$$

(CompL)

$$\frac{P \xrightarrow{\alpha} P' \qquad disjoint(P', Q)}{P \triangleleft_{\theta,M} Q \xrightarrow{\alpha} P' \triangleleft_{\theta,M} Q}$$

(CompR)

$$\frac{Q \xrightarrow{\alpha} Q' \qquad disjoint(P, Q') \qquad \alpha = \alpha' = \tau \ \vee\ (\alpha = a\#l(v) \ \wedge\ leftH(P)(M)(v) \Downarrow v' \ \wedge\ \alpha' = a\#l(v') \wedge \# \in \{!, ?\})}{P \triangleleft_{\theta,M} Q \xrightarrow{\alpha'} P \triangleleft_{\theta,M} Q'}$$

(CommSyncLR)

$$\frac{P \xrightarrow{a!v} P' \qquad Q \xrightarrow{a?v'} Q' \qquad leftH(P)(M^{-1})(v) \Downarrow v' \qquad disjoint(P', Q')}{P \triangleleft_{\theta,M} Q \xrightarrow{\tau} P' \triangleleft_{\theta,M} Q'}$$

(CommSyncRL)

$$\frac{Q \xrightarrow{a!v} Q' \qquad P \xrightarrow{a?v'} P' \qquad leftH(P)(M)(v) \Downarrow v' \qquad disjoint(P', Q')}{P \triangleleft_{\theta,M} Q \xrightarrow{\tau} P' \triangleleft_{\theta,M} Q'}$$

**Figure 3: Reduction rules for communication**

(Motion) rule checks the conditions of the motion primitives and makes sure the store matches the initial state and the final states of the trajectory. The (Time) rule puts together trajectories of processes. The trajectory of one process is projected to become part of the disturbances of the other process. More precisely, $\nu_P$ gets the external inputs projected on $P$ ($\nu|_P$) along the output of $Q$ through the connection ($\theta(\xi_Q)$) to which we apply the frame shift ($\xi(M^{-1})$). Finally, we also check that the resources used by the two processes stay disjoint during the execution of the motion primitives.

Finally, Figure 5 shows the semantics of the internal control flow of the processes. These are the standard rules for an imperative programming language. All these transitions are silent and follow the expected semantics of an imperative programming language.

At the root of the $\triangleleft_{\theta,M}$ tree, only $\xrightarrow{\tau}$ and $\xrightarrow{\xi,\nu,T}$ are allowed. Messages send and reception must be matched inside the system (closed world hypothesis). Once the send and receive are matched the label becomes $\tau$.

## 4 VERIFICATION

A PGCD program can "get stuck" during execution in different ways. First, the message passing can deadlock because send operations are blocking. Second, a process may try executing a motion primitive when its precondition does not hold. Third, two processes may execute motion primitives concurrently but their resources may intersect. In this section, we describe a verification algorithm for PGCD programs.

The key idea is that the programming model allows the verification problem to be separated into logical zero-time message-passing periods and real-time periods when time elapses following the trajectories defined by motion primitives. Our verification algorithm uses a combination of model-checking and constraint-solving. The model-checker checks the correctness of message communication between processes. A numerical solver for non-linear constraints over the reals checks the correctness of motion primitives.

*Communication safety.* For the messages, we want to show that the communication between components is well-formed. In particular, we verify that the program does not get into a state where some process is forever blocked on a send operation and that there is no unbounded execution solely with message passing (making time "stop").

Our verification algorithm converts a program into concurrently executing control flow automata (CFA) [13]. The code of each CFA is abstracted and we keep only send, receive, and motion primitives. Local computation is abstracted and internal choices (**if then else** ) becomes non-deterministic. Then, for each CFA, we check there is no loop (cycles in the CFA) without any motion primitive. This is a sufficient check to prevent potentially infinite "0-time" computation.

We take the synchronized product of all the CFAs. Matching **send** and **receive** statements in two processes synchronize based on labels. Motion primitives are considered to synchronize *all* processes globally in real time. Moreover, for **receive** statements, a motion primitive only executes when no more communication is possible. Finally, we check for deadlock, i.e., a non final state without successor, by exploring the state space of the synchronized product using a model checker. Rather than building the product explicitly, in our implementation, we encode the CFAs as Promela processes and perform the exploration using the Spin model checker [14]. Notice that due to the construction above there is only a single final state where all the processes have finished. Therefore, the deadlock check makes sure that either processes are communicating, executing a motion primitive, or terminated.

Furthermore, during the state space exploration, we also extract the combination of motion primitives that are executed concurrently. We use them in the second part of the verification process.

*Trajectories and footprints.* Using the combination of motion primitives recorded during model checking, we now check that the abstract motion primitives can execute correctly. This requires two checks. First, for motion primitives of different processes executed concurrently, we need to make sure that a trajectory satisfying the rules for motion (Figure 4) exist. Second, for motion primitives executed sequentially by the same process, we need to make sure that the post-condition of the first motion primitive implies the precondition of the following one. Currently we rely on the user giving state invariants in the form of program annotations to help with the verification process. These annotations associate predicates with program locations.

To check that motion primitives executing concurrently have a joint trajectory, we use an assume-guarantee style of reasoning.

(Motion)

$$\frac{\vee \begin{cases} S{=}\Sigma[\mathbf{m}] \wedge S'{=}\Sigma[\mathbf{skip}] \\ S{=}\Sigma[\mathbf{receive(m)}\{\ldots\}] \wedge S'{=}S \end{cases} \quad \xi(0) = \rho|_X \quad \rho' = \rho|_{Var} \cup \xi(T) \quad \mathbf{m}.\mathrm{Pre}(\xi(0), \nu(0)) \quad \mathbf{m}.\mathrm{Post}((\xi(T), \nu(T))) \quad \forall t \in [0,T].\ \mathbf{m}.\mathrm{Inv}((\xi(t), \nu(t)))}{(Var, \mathcal{M}, S, \rho, rsrc) \xrightarrow{\xi, \nu, T} (Var, \mathcal{M}, S', \rho', rsrc)}$$

(Time)

$$\frac{P \xrightarrow{\xi_P, \nu_P, T} P' \quad \nu_P = \nu|_P \cup \xi(M)(\theta(\xi_Q)) \quad Q \xrightarrow{\xi_Q, \nu_Q, T} Q' \quad \nu_Q = \nu|_Q \cup \xi(M^{-1})(\theta(\xi_P)) \quad \xi = (\xi_P \cup \xi_P(M)(\xi_Q)) \quad \forall t \in [0,T].\ P.rsrc(\xi_P(t)) \cap \xi_P(t)(M)(Q.rsrc(\xi_Q(t))) = \emptyset}{P \triangleleft_{\theta,M} Q \xrightarrow{\xi, \nu, T} P' \triangleleft_{\theta,M} Q'}$$

**Figure 4: Reduction rules for motion**

(Seq)

$$\frac{}{(Var, \mathcal{M}, \Sigma[\mathbf{skip}; S], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[S], \rho, rsrc)}$$

(Assign)

$$\frac{\rho(e) \Downarrow v \quad \rho' = \rho(x) \leftarrow v}{(Var, \mathcal{M}, \Sigma[x := e], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[\mathbf{skip}], \rho', rsrc)}$$

(WhileT)

$$\frac{\rho(e) \Downarrow true}{(Var, \mathcal{M}, \Sigma[\mathbf{while}\ e\ \mathbf{do}\ S], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[S; \mathbf{while}\ e\ \mathbf{do}\ S], \rho, rsrc)}$$

(WhileF)

$$\frac{\rho(e) \Downarrow false}{(Var, \mathcal{M}, \Sigma[\mathbf{while}\ e\ \mathbf{do}\ S], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[\mathbf{skip}], \rho, rsrc)}$$

(IteT)

$$\frac{\rho(e) \Downarrow true}{(Var, \mathcal{M}, \Sigma[\mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S'], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[S], \rho, rsrc)}$$

(IteF)

$$\frac{\rho(e) \Downarrow false}{(Var, \mathcal{M}, \Sigma[\mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S'], \rho, rsrc) \xrightarrow{\tau} (Var, \mathcal{M}, \Sigma[S'], \rho, rsrc)}$$

**Figure 5: Reduction rules for control flow.**

When two processes are attached, one process relies on the invariants of the other's output (which can be an external input) to satisfy its own invariant and vice versa.

The first step is to extract the assumptions and guarantees from the predicates. A predicate $\mathcal{P}$ of process $P$, e.g., the invariant of a motion primitive, is projected on to $P$'s external inputs to get the assumption and on $P$'s physical state variables to get the guarantee. That is, $A_{\mathcal{P}} \Leftrightarrow (\exists x \in P.X.\ \mathcal{P})$ and $G_{\mathcal{P}} \Leftrightarrow (\forall w \in P.W.\ \mathcal{P})$.

For assume-guarantee reasoning, we follow the method presented by Nuzzo [21]. Given a program $\Pi$ and an invariant $I_P$ for each process $P$, we first traverse $\Pi$ starting from the leaves to generate assume/guarantee contracts for each attached composition. Each $P \triangleleft_{\theta,M} Q$ gets a contract based on the contract of their children:

- $A \Leftrightarrow (A_P \vee M(A_Q) \vee \neg(G_P \wedge M(G_Q))) \wedge \forall (x,y) \in \theta.\ x = y$
- $G \Leftrightarrow G_P \wedge M(G_Q) \wedge \forall (x,y) \in \theta.\ x = y$

These rules are the composition rules from [21, Section 2.3.2] to which we have added the frame shifts. At each step of this process we need to check that the composed contracts are well-formed by checking:

- *compatibility*: $A$ is satisfiable,
- *consistency*: $G$ is satisfiable, and
- *spatial separation*: $G \Rightarrow P.rsrc \cap M(Q.rsrc) = \emptyset$ is valid.

The *spatial separation* check is new in our system and states that under the guarantees provided by both $P$ and $Q$, their respective resources must be disjoint. Furthermore, when the root of $\Pi$ is reached, the final $A$ must be implied by the environment assumption and the final $G$ is the overall behavior of the system.

*Specifications and Annotations.* The verifier for PGCD is not fully automatic; to help the verifier, the programmer needs to provide some annotations. As mentioned earlier, a motion primitive **m** is specified by (Pre, Inv, Post). Further, a resource function for a robot with a complicated geometry can be complex; for example, for a robotic arm, it can be the geometry of the arm itself. In practice, the check for spatial separation uses programmer-specified *abstract footprint* predicates which over-approximate $P.rsrc$ and $Q.rsrc$ but for which the separation check is efficient.

For instance, in the **grab** motion primitive of an arm, we can over-approximate the arm's working envelope (which can be a complex and non-convex set) by a half sphere around its base with a radius corresponding to the arm's maximal extension. This formula is simpler than the arm's resource function as it does not depend on the arm's state but may be sufficient to handle some scenarios. With the extra footprint specification, it becomes possible to divide the collision check in two parts: (1) we check that each component stays within the footprint of its motion primitive and (2) we check that the footprint of concurrently executing motion primitives do not intersect.

The second type of user annotations are invariants given as constraints over the system's state at particular program locations. For instance, in Algorithms 1 and 2 when the cart is a line 6 and the arm at line 2, we have $|target - C.p| \leq target \wedge A.\gamma = 0 \wedge A.\beta = minLowerAngle \wedge A.\alpha = maxUpperAngle$. This is the conjunction of the exiting the loop (Algorithms 1 line 4) and the arm in a folded state. These invariants have a role similar to loop invariants and allow us to decompose the reasoning into a finite number of checks.

# 5 IMPLEMENTATION AND EVALUATION

## 5.1 Implementation

We have implemented PGCD and analyses as an interpreter, library, runtime system, and verifier in Python. The code is publicly available at https://github.com/MPI-SWS/pgcd. To run a program, each robot runs a copy of the interpreter and the runtime. Each copy of the interpreter runs the processes on actual robot hardware, directly interacting with the hardware and communicating through

the runtime system. The runtime system manages the messages using an additional server acting as a central broker that keeps up-to-date frame shifts between the different robots.

The verifier takes as input a program (with invariant annotations) and specifications for the motion primitives and the environment. Then, it decomposes the program into a list of processes, their connections, and respective specifications. Finally, it performs the checks described in Section 4.

*Runtime System.* We use ROS for the message-passing layer [23]. ROS is a publish-subscribe system where processes advertise topics, publish messages or subscribe on specific topics. Topics can be hierarchically arranged in namespaces. A ROS master node manages the topics.

In our runtime system, each process get a namespace based on its identity, and the topics correspond to the labels of the messages. While ROS implements asynchronous message-passing by default, we implement a synchronous rendezvous communication on top to faithfully model our semantics. When a process encounters a **receive** statement, it subscribes to the topics corresponding to the labels occurring in the receive block. After receiving a message, it unsubscribes from the topics. This enables the sender to query the ROS framework to check the presence of a receiver and to block until a receiver is ready. Therefore, the send operation blocks, accurately implementing our synchronous semantics.

The use of ROS lets us reuse a lot of robotics infrastructure available in the ROS ecosystem. We build on ROS's TF2 library [10] to deal with frame shifts. In our implementation, every component periodically publishes its frame shift relative to their children. (Periodic updates of the frame shift are necessary as frame shifts are dynamic, i.e., based on the current state.) The receive operation for a process queries TF2 for the frame shift from the sender's frame to the frame of the recipient process and transforms the content of the message appropriately.

Motion primitives are hardware specific and each motion primitive is currently implemented directly for the corresponding robot. The implementation of motion primitives interfaces with the hardware (e.g., to motors) for individual robots.

*Verifier.* Our verifier is also implemented in PYTHON. To check the communication, we generate PROMELA models from the program and analyze them with SPIN [14]. For the geometric reasoning we use SYMPY, a symbolic manipulation package which contains a module for 3D Cartesian coordinate systems.[1] Using this module, formulas are expressed in their component's frame and frames are constructed on top of their parent. When the frame shifts are given, a formula can be automatically translated to a specific coordinate system using symbolic manipulation in SYMPY. As the frame shift and motion primitives involve reasoning about non-linear arithmetic, e.g., trigonometric functions arising from rotations, we use the DREAL solver [11] for non-linear theories over the real to discharge the verification conditions.

Each motion primitive needs to have a specification in the form of a (Pre, Inv, Post) triple. The specifications are written directly as PYTHON functions by extending the appropriate base classes provided by the tool. The specification of a motion primitive describes "frame conditions" by explicitly describing the variables modified (using a **modifies** clause similar to ESC/Java [9]). Additionally, we allow the programmer to separately specify footprints (regions of space used by the motion primitive) for Pre, Inv, and Post. For example, the resource function of an arm consists of a number of cylinders with spherical joints, but the footprint can simply return a half-sphere bounding the arm.

The verifier checks that the names of messages and motion primitives coincide, and the values passed to the motion primitives syntactically match those in the specification. Thus, correct programs, which may perform local computations to provide values to the motion primitives, may be rejected. Verifying programs in the presence of local computations would require implementing a symbolic execution engine. The stronger syntactic check was sufficient for our examples.

## 5.2 Experiments

We have implemented several examples involving multi-robot coordination in our system. First, we describe our experimental setup, both for the hardware and software. Then, we explain the experiments. Finally, we report on the size of the programs, specifications, and verification time. [2]

*Setup.* We built a robotic arm and two carts, shown in Figure 1, using a mix of off-the-self parts and customized 3D printed components. The arm is based on the BCN3D MOVEO,[3] where the upper arm section is shortened to make it lighter and easier to mount. The two carts are omnidirectional driving platforms. They have three degrees of freedom (two in translation, one in rotation) when moving on a flat ground. The advantage of using such wheels is that all the three degrees of freedom are controllable and movement does not require complex planning. The smaller cart is battery powered and the larger one has a tether due to the large power consumption of the arm mounted on top.

The carts and the arm use stepper motors to precisely control the motion. The carts do not have feedback on their position and keep track of their state using *dead reckoning*, i.e., they know their initial state and then they update their virtual state by counting the number of steps the motors make. If we control slippage and do not exceed the maximum torque of the motors, there is little accumulation of error as long as the initial state is correct.[4] Furthermore, using stepper motors allows us to know the time it takes to execute a given motion primitive by fixing the rate of steps.

Each robot has a RaspberryPi 3 model B running the process and the runtime system. The ROS master node, providing the messaging services, runs on a separate RaspberryPi to which all the processes connect. The RaspberryPi runs Raspbian OS (based on Debian Jessie) with ROS Kinetic Kame.

The carts each have five motion primitives, the arm has eight. Motion primitives for the carts require 92 and 97 lines of code, the arm has 154 lines, and there is an additional 151 lines of shared code. The specification consists of 174 and 75 lines for the carts

[1] http://docs.sympy.org/latest/modules/vector/index.html

[2] A short video of our experiments can be seen at https://owncloud.mpi-sws.org/index.php/s/0olpAuC7nq6wKTJ/download?path=%2F&files=All%20Experiments.mp4.

[3] https://github.com/BCN3D/BCN3D-Moveo

[4] In our experiments, we use markings on the ground to fix the initial state as can be seen in Figure 6 and the video.

and 221 lines for the arm. In Table 1, we give an estimate of the implementation and specification effort for motion primitives in terms of lines of code (Loc). The two carts are on the same line as they share much of the same specification. However, the implementation of the motion primitives differ due to the specific hardware configurations of the two robots. The *Shared* line represents the shared code base that interfaces with the hardware.

*Experiments.* We tested our tools on four experiments. In the following, we call the smaller cart the (object) carrier.

**Fetch.** This is the running example.

**Handover.** This experiment is a variation of "fetch." We added an extra carrier cart which carries the object. The two carts meet before the arm takes an object placed on top of the carrier and, then, they go back to their initial position (see Figure 6b).

**Twist and Turn.** In this experiment, the carrier starts in front of the cart. The arm takes an object from the carrier. Then all three robots moves simultaneously. The cart rotates in place, the carrier describe a curve around the cart, and arm move from on side of the cart to the other side. At the end, the arm puts the object back on the carrier. This can be seen in Figure 6c.

**Underpass.** In this experiment, the carrier cart brings an object to the arm which is then taken by the arm. The carrier cart goes around the arm passing under an obstacle which is high enough for just the carrier but not with the object on top. Finally, the arm puts the object back on the carrier on the other side of the obstacle. This can be seen in Figure 6d.

Composite images (combination of multiple frame of the video) are shown in Figure 6. The carts implement motion explicitly using the motion primitives (move straight, strafe, rotate, sweep). For instance, when going around the cart in the last experiment, the carrier executes rotate, move straight, rotate, strafe. Because gripping is a collision, we exclude the gripper from the arm's footprint and we do not model the objects we grip. For the environment, we model obstacles as regions of $\mathbb{R}^3$ and also test for collision against these regions.

Table 2 shows the size of the programs in the core language of Section 3 (sum for all the robots) and the size of the specifications. The program includes the statements for each process and the connections between processes. As part of the program we include the "world" description: the world is a virtual process which is the root of the parent-child attached composition. Additionally, the world contains obstacles used for additional collision checks. Finally, we show the number of verification conditions (#VCs) generated when checking the motion primitives and the total running time. In all cases, the communication protocol are simple and are verified by Spin in less than a second. Checking the motion primitives require more complex reasoning and dominate the running time. The total number of verification conditions is quite large as showing the absence of collision is quadratic in the number of components.

Overall, the experiments show that PGCD is expressive enough for complex coordination tasks and, at the same time, the verifier can scale to statically verify concurrency and geometric properties of these tasks.

# 6 LIMITATIONS AND FUTURE WORK

We now discuss some limitations of PGCD and potential future directions.

To verify the coordination, we rely on model checking the global system with all the components. While our current examples have simple enough communication protocols and Spin can exhaustively check that the communication protocol is well formed, this solution is not completely satisfactory.

(1) To keep the state space small, we currently abstract details of the computations and values inside the messages. This leads to a loss of precision and imposes a higher annotation burden.
(2) The model checking works on the global system and does not take advantage of the structure of the composition. In many practical cases, we expect the communication to follow the physical structure of the system. For instance, the composed cart and arm have internal communication for synchronization but, from the outside, they behave like a single entity. The cart and arm assembly responds to requests from other processes while keeping their internal communication hidden.

To improve these points, we are investigating the use of choreographic programming [19] and multiparty session types [4, 15]. These methods use global specification structured to be projectable on the individual processes. Then, the verification only requires local checks at the level of each process.

For motion primitives, most of the checks we perform are related to the disjointness of footprints, i.e., that different components do not collide. Currently, we encode these constraints in first order logic. We plan to embed our programs into an expressive logic for resources [24] to deal with these constraints more efficiently.

We are also exploring simultaneous concurrent programming and distributed controller synthesis. As an example, assume that we have two cart and arm compositions which should lift one object together. Suppose that that lifting the object with only one arm would cause the cart/arm composition to tilt over. Thus, there is a strong coupling between all components during the coordinated lift of the object. Our programming model is expressive enough to capture the synchronization. However, a robust controller in this setting would need (almost) continuous feedback between all components to fulfill the coordinated lift task. Thus, our model of loosely coupled motion primitives, one per component, would be too weak.

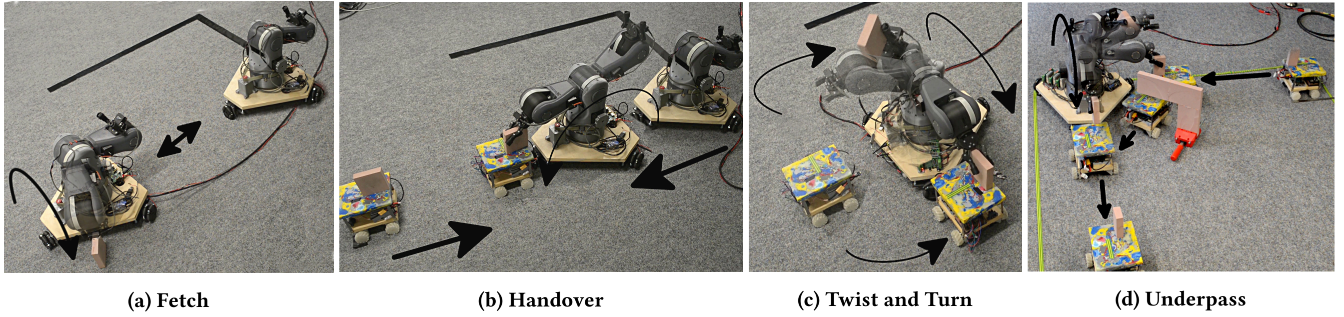
# 7 CONCLUSION

We have presented a language and verification system for concurrent, communicating components interacting with the physical world and embedded in geometric space. The semantics of the language takes into account relative frames of reference among the components and transforms geometric data in communication to the appropriate frame.

Our evaluation demonstrates that our language and runtime can specify, verify, and run non-trivial co-ordinations between multiple robots in reasonable programming effort while providing automated verification support.

**Table 1: Motion Primitives Implementation and Specification**

| Robot | Motion Primitives | Implementation (LoC) | Specification (LoC) |
|---|---|---|---|
| Arm | SetTurntable, SetLowerArm, SetUpperArm, SetPose, Grab, Grip, Fold, Idle | 154 | 221 |
| Cart (both) | Move, Strafe, Rotate, Sweep, Idle | 92 + 97 | 174 + 75 |
| *Shared* | – | 151 | – |



**(a) Fetch** **(b) Handover** **(c) Twist and Turn** **(d) Underpass**

**Figure 6: Composite images of the experiments.**

**Table 2: Programs, Annotations, and Checks**

| Experiment | Program (LoC) | Annotations (LoC) | #VCs | Time (sec.) |
|---|---|---|---|---|
| Fetch | 35 | 12 | 82 | 16 |
| Handover | 29 | 18 | 183 | 86 |
| Twist and Turn | 38 | 18 | 93 | 79 |
| Underpass | 52 | 40 | 393 | 103 |

## ACKNOWLEDGMENTS

This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

## REFERENCES

[1] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. 2006. Compositional modeling and refinement for hierarchical hybrid systems. *J. Log. Algebr. Program.* 68, 1-2 (2006), 105–128.

[2] Rajeev Alur and Thomas A. Henzinger. 1997. Modularity for Timed and Hybrid Systems. In *CONCUR (LNCS)*, Vol. 1243. Springer, 74–88.

[3] J.A. Bergstra and C.A. Middelburg. 2005. Process algebra for hybrid systems. *Theoretical Computer Science* 335, 2 (2005), 215 – 280. Process Algebra.

[4] L. Bocchi, W. Yang, and N. Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR 2014 (LNCS)*, Vol. 8704. Springer, 419–434.

[5] Joseph Campbell, Cumhur Erkan Tuncali, Peng Liu, Theodore P. Pavlic, Ümit Özgüner, and Georgios E. Fainekos. 2016. Modeling concurrency and reconfiguration in vehicular systems: A $\pi$-calculus approach. In *CASE*. IEEE, 523–530.

[6] Luca Cardelli and Philippa Gardner. 2012. Processes in space. *Theor. Comp. Sci.* 431 (2012), 40–55.

[7] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. 2016. Model Checking Spatial Logics for Closure Spaces. *Logical Methods in Computer Science* 12, 4 (2016).

[8] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2017. DRONA: a framework for safe distributed mobile robotics. In *ICCPS 17*. ACM, 239–248.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *PLDI*. ACM, 234–245.

[10] Tully Foote. 2013. tf: The transform library. In *(TePRA) Technologies for Practical Robot Applications (Open-Source Software workshop)*. 1–6.

[11] Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *CADE-24 (LNCS)*, Vol. 7898. Springer, 208–214.

[12] Ritwika Ghosh, Sasa Misailovic, and Sayan Mitra. 2018. Language Semantics Driven Design and Formal Analysis for Distributed Cyber-Physical Systems: [Extended Abstract]. In *ApPLIED@PODC 2018*. ACM, 41–44.

[13] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *POPL*, John Launchbury and John C. Mitchell (Eds.). ACM, 58–70.

[14] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67.

[16] Ruggero Lanotte and Massimo Merro. 2017. A Calculus of Cyber-Physical Systems. In *LATA*. Springer, 115–127.

[17] Steven M. LaValle. 2006. *Planning algorithms*. Cambridge University Press.

[18] Yixiao Lin and Sayan Mitra. 2015. StarL: Towards a Unified Framework for Programming, Simulating and Verifying Distributed Robotic Systems. In *LCTES 15*. ACM, 9:1–9:10.

[19] Hugo A. López and Kai Heussen. 2017. Choreographing Cyber-physical Distributed Control Systems for the Energy Sector. In *Proceedings of the Symposium on Applied Computing (SAC '17)*. ACM, New York, NY, USA, 437–443.

[20] Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. 2003. Hybrid I/O automata. *Inf. Comput.* 185, 1 (2003), 105–157.

[21] Pierluigi Nuzzo. 2015. *Compositional Design of Cyber-Physical Systems Using Contracts*. Ph.D. Dissertation. EECS Department, UC Berkeley.

[22] André Platzer. 2010. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer.

[23] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*.

[24] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74.

[25] William C. Rounds and Hosung Song. 2003. The Phi-Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems. In *HSCC*. Springer, 435–449.