Motion Session Types for Robotic Interactions

Rupak Majumdar
MPI-SWS, Saarbrücken, Germany
rupak@mpi-sws.org

Marcus Pirron
MPI-SWS, Saarbrücken, Germany
mpirron@mpi-sws.org

Nobuko Yoshida
Imperial College London, UK
n.yoshida@imperial.ac.uk

Damien Zufferey
MPI-SWS, Saarbrücken, Germany
zufferey@mpi-sws.org

Abstract

Robotics applications involve programming concurrent components synchronising through messages while simultaneously executing *motion primitives* that control the state of the physical world. Today, these applications are typically programmed in low-level imperative programming languages which provide little support for abstraction or reasoning.

We present a unifying programming model for concurrent message-passing systems that additionally control the evolution of physical state variables, together with a compositional reasoning framework based on multiparty session types. Our programming model combines *message-passing concurrent processes* with *motion primitives*. Processes represent autonomous components in a robotic assembly, such as a cart or a robotic arm, and they synchronise via discrete messages as well as via motion primitives. Continuous evolution of trajectories under the action of controllers is also modelled by motion primitives, which operate in global, physical time.

We use multiparty session types as specifications to orchestrate discrete message-passing concurrency and continuous flow of trajectories. A global session type specifies the communication protocol among the components with joint motion primitives. A projection from a global type ensures that jointly executed actions at end-points are *communication safe* and *deadlock-free*, i.e., session-typed components do not get stuck. Together, these checks provide a compositional verification methodology for assemblies of robotic components with respect to concurrency invariants such as a progress property of communications as well as dynamic invariants such as absence of collision.

We have implemented our core language and, through initial experiments, have shown how multiparty session types can be used to specify and compositionally verify robotic systems implemented on top of off-the-shelf and custom hardware using standard robotics application libraries.

2012 ACM Subject Classification Computer systems organization → Robotics; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi; Theory of computation → Type theory

Keywords and phrases Session Types, Robotics, Concurrent Programming, Motions, Communications, Multiparty Session Types, Deadlock Freedom

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.28

Category Brave New Idea Paper

Funding *Rupak Majumdar*: DFG 389792660 TRR 248, ERC Synergy Grant 610150.
Marcus Pirron: DFG 389792660 TRR 248, ERC Synergy Grant 610150.
Nobuko Yoshida: EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201/1.
Damien Zufferey: DFG 389792660 TRR 248, ERC Synergy Grant 610150.

1 Introduction

Many cyber-physical systems today involve an interaction among communication-centric components which together control trajectories of physical variables. For example, consider an autonomous robotic system executing in an assembly line. The components in such an example would be robotic manipulators or arms as well as robotic carts onto which one or more arms may be mounted. A global task may involve communication between the carts and the arms – for example, to jointly decide the position of the arms and to jointly plan trajectories – as well as the execution of motion primitives – for example, to follow a trajectory or to grip an object. Today, a programmer developing such an application must manually orchestrate the messaging and the dynamics: errors in either can lead to potentially catastrophic system failures. Typically, programs are written in (untyped) imperative programming language using messaging libraries. Arguments about correctness are informal at best, with no support from the language.

In this paper, we take the first steps towards a uniform programming model for autonomous robotic systems. Our model combines message-based communication with physical dynamics ("motion primitives") over time. Our starting point is the notion of *multiparty session types* [25, 26, 10], a principled, type-based, discipline to specify and reason about *global communication protocols* in a concurrent system. We enrich a process-based core language for communication with the ability to execute *dynamic motion primitives* over time. Motion primitives encapsulate the actions of dynamic controllers on the physical world and define the continuous evolution of the trajectories of the system. At the same time, we enrich a type system for multiparty session-based communication with motion primitives.

The interaction of communication and dynamics is non-trivial. Since time is global to a physical system, every independently running process must be ready to execute their motion primitives simultaneously. Thus, for example, programs in which one component is blocked waiting for a message while another moves along a trajectory must be ruled out as ill-typed. To keep the complexity of the problem manageable, our semantics keeps, as much as possible, the message exchanges separate from the continuous trajectories. In particular, in our model, message exchanges occur instantaneously and at discrete time steps, à la synchronous reactive programming, while motion primitives execute in global time. System evolution is then organised into rounds; each round consists of a logical time for communication followed by physical time for motion. This assumption is realistic for systems where the speed of the trajectories is comparatively slow compared to the message transmission delay.

Our reasoning principles closely follow the usual type-checking approach of multiparty session types. Specifications are described through *global types*, which constrain both message sequences and motion sequences. Global types are projected to *local types*, which specify the actions in a session from the perspective of a single end-point process. Finally, a verification step checks that each process satisfies its local type. The soundness theorem ensures that in this last case, the composition of the processes satisfy a protocol compliance.

Our type system ensures communication safety and deadlock-freedom for messages, ensuring, for example, that communication is not stuck or time cannot progress. In addition, we verify safety properties of physical trajectories such as non-collision by constraint-based verification of simultaneously executed motion primitives specified in the global type.

Existing session type formalisms such as [9] fall short to model a combination of individual interactions and global synchronisations by motions. To demonstrate our initial step and to observe an effect of new primitives specific to robotics interactions, we start from the simplest multiparty session type system in [15, 18]. The programming model and type

system introduced in this paper provides the foundations for PGCD programs, a practical programming system to develop concurrent robotics applications [4]. We have used our calculus and type system to verify correctness properties of (abstract versions of) multi-robot co-ordination programs written in PGCD, which then execute on real robotics hardware. Our evaluation shows that multiparty session types and choreographies for multi-robot co-ordination and manipulation can lead to statically verified implementations that run on off-the-shelf and custom robotics hardware platforms.

Outline. We first give a gentle introduction to motion session types to those who are interested in concurrent robotics programming, but not familiar with session types. Section 3 discusses a core abstract calculus of processes where motions are abstracted by just the passage of time; Section 4 defines a typing system with motion primitives; Section 5 extends our theory to deal with continuous trajectories; Section 6 discusses our implementation; Section 7 gives related work and Section 8 concludes.

2 A Gentle Introduction to Motion Session Types

The aim of this section is to give a gentle introduction of motion session types for readers who are interested in robotics programming but who are not familiar with session types nor process calculi.

A key difficulty in robotics programming is that the programmer has to reason about concurrent processes communicating through messages as well as about dynamics evolving in time. The idea of motion session types is to provide a typing framework to only allow programs that follow structured sequences of interactions and motion. A *session* will be a natural unit of structured communication and motion. *Motion session types* abstract the structure of a session. and provide a syntax-driven approach to restricting programs to a well-behaved subclass – for this subclass, one can check processes compositionally and derive properties of the composition.

Motion session types extend session types, introduced in a series of papers during the 1990s [23, 43, 24], in the context of pure concurrent programming. Session types have since been studied in many contexts over the last decade – see the surveys of the field [27, 17].

We begin by an overview of the key technical ideas of multiparty session types. Then we introduce motion primitives to multiparty session types for specifying actions over time. Finally, we refine the motion primitives to physical motion executed by the robots.

2.1 Communication: Multiparty Session Types

We begin with a review of multiparty session types, a methodology to enable compositional reasoning about communication.

As a simple example, consider a scenario in which a cart and arm assembly has to fetch objects. We associate a process with each physical component; thus, we model the scenario using a *cart* (Cart) and an *arm* (Arm) attached to the cart. The task involves synchronisation between the cart and the arm as well as co-ordinated motion. Synchronization is obtained through the exchange of messages. We defer the discussion on motion to Section 2.2.

Specifically, the protocol works as follows.

1. The cart sends the arm a fold command *fold*. On receiving the command, the arm folds itself. When the arm is completely folded, it sends back a message *ok* to the cart. On receipt of this message, the cart moves.

2. When the cart reaches the object, it stops and sends a *grab* message to the arm to grab the object. While the cart waits, the arm executes the grabbing operation, followed by a folding operation. Then the arm sends a message *ok* to the cart. This sequence may need to be repeated.
3. When all tasks are finished, the cart sends a message *done* to the arm, and the protocol terminates.

The multiparty session types methodology is as follows. First, define a *global type* that gives a shared contract of the allowed pattern of message exchanges in the system. Second, *project* the global type to each end-point participant to get a *local type*: an obligation on the message sends and receipts for each process that together ensure that the pattern of messages are allowed by the global type. Finally, check that the implementation of each process conforms to its local type.

In our protocol, from a global perspective, we expect to see the following pattern of message exchanges, encoded as a *global type* for the communication:

$$\mu\mathbf{t}.\mathsf{Cart} \to \mathsf{Arm}:\{\mathit{fold}.\mathsf{Arm} \to \mathsf{Cart}:\mathit{ok}.\mathsf{Cart} \to \mathsf{Arm}:\mathit{grab}.\mathsf{Arm} \to \mathsf{Cart}:\mathit{ok}.\mathbf{t},\ \mathit{done}.\mathtt{end}\} \quad (1)$$

The type describes the global pattern of communication between Cart and Arm using message exchanges, sequencing, choice, and repetition. The basic pattern $\mathsf{Cart} \to \mathsf{Arm}:m$ indicates a message m sent from the Cart to the Arm. The communication starts with the cart sending either a *fold* or a *done* command to the arm. In case of *done*, the protocol ends (type `end`); otherwise, the communication continues with the sequence *ok*. *grab*. *ok* followed by a repetition of the entire pattern. The operator "." denotes sequencing, and the type $\mu\mathbf{t}.T$ denotes recursion of T.

The global type states what are the valid message sequences allowed in the system. When we implement Cart and Arm separately, we would like to check that their composition conforms to the global type. We can perform this check compositionally as follows.

Since there are only two participants, projecting to each participant is simple. From the perspective of the Cart, the communication can be described by the type:

$$\mu\mathbf{t}.((\ !\mathit{fold}.\ ?\mathit{ok}.\ !\mathit{grab}.\ ?\mathit{ok}.\mathbf{t}\ ) \quad \oplus \quad (\ !\mathit{done}.\ \mathtt{end}\ )) \quad (2)$$

where $!m$ denotes a message m sent (to the Arm) and $?m$ denotes a message m received from the Arm. and $\oplus$ denotes an (internal) choice. Thus, the type states that Cart repeats actions $!\mathit{fold}.\ ?\mathit{ok}.\ !\mathit{grab}.\ ?\mathit{ok}$ until at some point it sends *done* and exits.

Dually, from the viewpoint of the Arm, the same global session is described by the dual type

$$\mu\mathbf{t}.((\ \ ?\mathit{fold}.\ !\mathit{ok}.\ ?\mathit{grab}.\ !\mathit{ok}.\mathbf{t}\ ) \quad \& \quad (\ ?\mathit{done}.\ \mathtt{end}\ )) \quad (3)$$

in which & means that a choice is offered externally.

We can now individually check that the implementations of the cart and the arm conform to these local types.

The global type seems overkill if there are only two participants; indeed, the global type is uniquely determined given the local type (2) or its dual (3). However, for applications involving *multiple parties*, the global type and its projection to each participant are essential to provide a shared contract among all participants.

For example, consider a simple ring protocol, where the Arm process above is divided into two parts, Lower and Upper. Now, Cart sends a message *fold* to the lower arm Lower, which forwards the message to Upper. After receiving the message, Upper sends an acknowledgement *ok* to Cart. We start by specifying the global type as:

$$\mathsf{Cart} \to \mathsf{Lower} : \mathit{fold}.\mathsf{Lower} \to \mathsf{Upper} : \mathit{fold}.\mathsf{Upper} \to \mathsf{Cart} : \mathit{ok}.\mathtt{end} \tag{4}$$

As before, we want to check each process locally against a local type such that if each process conforms to its local type then the composition satisfies the global type.

The global type in (4) is *projected* into the three endpoint session types:

Cart's endpoint type:	$\mathsf{Lower}!\mathit{fold}.\mathsf{Upper}?\mathit{ok}.\mathtt{end}$
Lower's endpoint type:	$\mathsf{Cart}?\mathit{fold}.\mathsf{Upper}!\mathit{fold}.\mathtt{end}$
Upper's endpoint type:	$\mathsf{Lower}?\mathit{fold}.\mathsf{Cart}!\mathit{ok}.\mathtt{end}$

where $\mathsf{Lower}!\mathit{fold}$ means "send to Lower a *fold* message," and $\mathsf{Upper}?\mathit{ok}$ means "receive from Upper an *ok* message." Then each process is type-checked against its own endpoint type. When the three processes are executed, their interactions automatically follow the stipulated scenario.

If instead of a global type, we only used three separate binary session types to describe the message exchanges between Cart and Lower, between Lower and Upper, and between Upper and Cart, respectively, without using a global type, then we lose essential sequencing information in this interaction scenario. Consequently, we can no longer guarantee deadlock-freedom among these three parties. Since the three separate binary sessions can be interleaved freely, an implementation of the Cart that conforms to $\mathsf{Upper}?\mathit{ok}.\mathsf{Lower}!\mathit{fold}.\mathtt{end}$ becomes typable. This causes the situation that each of the three parties blocks indefinitely while waiting for a message to be delivered. Thus, we shall use the power of multiparty session types to ensure correct communication patterns.

2.2 Motion: Motion Primitives and Trajectories

So far, we focused on the communication pattern and ignored the physical actions of the robots. Our framework of motion session types extends multiparty session types to also reason about *motion primitives*, which model change of state in the physical world effected by the robots. We add motion in two steps: first we treat motion primitives as abstract actions that have associated durations, and second as dynamic trajectories.

Abstractly, we model motion primitives as actions that take physical time. Accordingly, we extend session types with motion primitive $\mathsf{dt}\langle \mathsf{p}_i : a_i \rangle$, which indicates that the participants p_i jointly execute motion primitives a_i for the same duration of time.

Let us add the motion primitives to the cart and arm example. Recall that on receiving the command *fold*, the arm folds itself; meanwhile, the cart waits. When the arm is completely folded, it sends back a message to the cart, then the cart moves, following a trajectory to the object. This means the time the arm folds and the time the cart is idle (waiting for the arm) should be the same. Similarly, the time cart is moving and the idle time the arm waits for the cart should be synchronised. This explicit synchronisation is represented by the following global type:

$$\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}.\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold} \rangle.$$
$$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}.\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle} \rangle.G$$

where "$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle$" specifies the joint motion primitives idle executed by the Cart and fold executed by the Arm are synchronised. We extend local types with motion primitives as well. The conformance check ensures that, if each process conforms to its local types, then the composition of the system conforms to the global type – which now includes both message-based synchronization as well as synchronization over time using motion primitives.

Finally, we expand the abstract motion primitives with the underlying dynamic controllers and ensure that the joint execution of motion primitives is possible in the system. This requires refining each motion primitive to its underlying dynamical system and checking that whenever the global type specifies a joint execution of motion primitives, there is in fact a joint trajectory of the system that can be executed.

3 Motion Session Calculus

We now introduce the syntax and semantics of a synchronous multiparty motion session calculus. Our starting point is to associate a process with the physical component it controls. This can be either a "complete" robot or parts of a robot (like the cart or arm in the previous section). This makes it possible to model modular robots where parts may be swapped for different tasks. In the following, we simply say "robot" to describe a physical component (which may be a complete robot or part of a larger robot). Our programming model will associate a process with each such robot.

We build our motion session calculus based on a session calculus studied in [15, 18], which simplifies the synchronous multiparty session calculus in [29] by eliminating both shared channels for session initiations and session channels for communications inside sessions.

▶ **Notation 3.1** (Base sets). *We use the following base sets:* values*, ranged over by* $\mathsf{v}, \mathsf{v}', \ldots$*;* expressions*, ranged over by* $\mathsf{e}, \mathsf{e}', \ldots$*;* expression variables*, ranged over by* $x, y, z \ldots$*;* labels*, ranged over by* $\ell, \ell', \ldots$*;* session participants*, ranged over by* $\mathsf{p}, \mathsf{q}, \ldots$*;* motion primitives*, ranged over by* a*,* b*,* $\ldots$*;* process variables*, ranged over by* $X, Y, \ldots$*;* processes*, ranged over by* $P, Q, \ldots$*; and* multiparty sessions*, ranged over by* $M, M', \ldots$*.*

Motion Primitives

When reasoning about communication and synchronisation, the actual trajectory of the system is not important and only the time taken by a motion is important. Therefore, we first abstract away trajectories by just keeping the name of the motion primitive (a, b, ...) and, for each motion, we assume we know up front how long the action takes. We use the notation $\mathsf{dt}\langle a\rangle$ to represent that a motion primitive executes and time elapses. Every motion can have a different, a priori known, duration denoted $\mathit{duration}(a)$. We write the tuple $\mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle$ to denote a group of processes executing their respective motion primitives at the same time. For the sake of simplicity, we sometimes use a for both single or grouped motions. In Section 5, we look in more details into the trajectories defined by the joint execution of motion primitives.

Syntax of Motion Session Calculus

A value v can be a natural number n, an integer i, a Boolean true / false, or a real number. An expression e can be a variable, a value, or a term built from expressions by applying (type-correct) computable operators. The processes of the synchronous multiparty session calculus are defined by:

$$P \quad ::= \quad \mathsf{p}!\ell\langle \mathsf{e}\rangle.P \;\mid\; \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i \;\mid\; \sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.P \;\mid\; \mathsf{dt}\langle a\rangle.P$$
$$\mid \quad \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ P \;\mid\; \mu X.P \;\mid\; X \;\mid\; \mathbf{0}$$

The output process $\mathsf{p}!\ell\langle \mathsf{e}\rangle.Q$ sends the value of expression e with label ℓ to participant p. The sum of input processes (external choice) $\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i$ is a process that can accept a value with label ℓ_i from participant p for any $i \in I$; $\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.P$ is an external choice with a *default branch* with a motion action $\mathsf{dt}\langle a\rangle.P$ which can always proceed when there is no message to receive. According to the label ℓ_i of the received value, the variable x_i is instantiated with the value in the continuation process P_i. We assume that the set I is always finite and non-empty. The conditional process $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ represents the internal choice between processes P and Q. Which branch of the conditional process will be taken depends on the evaluation of the expression e. The process $\mu X.P$ is a recursive process. We assume that the recursive processes are *guarded*. For example, $\mu X.\mathsf{p}?\ell(x).X$ is a valid process, while $\mu X.X$ is not. We often omit $\mathbf{0}$ from the tail of processes.

We define a *multiparty session* as a parallel composition of pairs (denoted by $\mathsf{p} \triangleleft P$) of participants and processes:

$$M \quad ::= \quad \mathsf{p} \triangleleft P \;\mid\; M \mid M$$

with the intuition that process P plays the role of participant p, and can interact with other processes playing other roles in M. The participants correspond to the physical components in the system and the processes correspond to the code run by that physical component. A multiparty session is *well formed* if all its participants are different. We consider only well-formed multiparty sessions.

Operational Semantics of Motion Session Calculus

The value v *of expression* e (notation $\mathsf{e} \downarrow \mathsf{v}$) is computed as expected. We assume that $\mathsf{e} \downarrow \mathsf{v}$ is effectively computable and takes logical "zero time."

We adopt some standard conventions regarding the syntax of processes and sessions. Namely, we will use $\prod_{i\in I} \mathsf{p}_i \triangleleft P_i$ as short for $\mathsf{p}_1 \triangleleft P_1 \mid \ldots \mid \mathsf{p}_n \triangleleft P_n$, where $I = \{1, \ldots, n\}$. We will sometimes use infix notation for external choice process. For example, instead of $\sum_{i\in\{1,2\}} \mathsf{p}?\ell_i(x).P_i$, we will write $\mathsf{p}?\ell_1(x).P_1 + \mathsf{p}?\ell_2(x).P_2$.

The *computational rules of multiparty sessions* are given in Table 1. They are closed with respect to structural congruence. The structural congruence includes a recursion rule $\mu X.P \equiv P\{\mu X.P/X\}$, as well as expected rules for multiparty sessions such as $P \equiv Q \Rightarrow \mathsf{p} \triangleleft P \mid M \equiv \mathsf{p} \triangleleft Q \mid M$. Other rules are standard from [15, 18]. However, unlike the usual treatment of π-calculi, our structural congruence does not have a rule to simplify inactive processes ($\mathsf{p} \triangleleft \mathbf{0}$). The reason is that even when a program might be logically terminated, the physical robot continues to exist and may still collide with another robot. Therefore, in our model, all processes need to terminate at the same time, and so we need to keep $\mathsf{p} \triangleleft \mathbf{0}$.

In rule [COMM], the participant q sends the value v choosing the label ℓ_j to participant p, who offers inputs on all labels ℓ_i with $i \in I$. In rules [T-CONDITIONAL] and [F-CONDITIONAL], the participant p chooses to continue as P if the condition e evaluates to true and as Q if e evaluates to false. Rule [R-STRUCT] states that the reduction relation is closed with respect to structural congruence. We use $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$.

The motion primitives are handled with [MOTION] and [M-PAR]. Here, we need to label transitions with the time taken by the action and propagate these labels with the parallel composition. This ensures that when (physical) time elapses for one process, it elapses

Table 1 Reduction rules. The communication between an output and an external choice (without the default motion action) is formalised similarly to [COMM].

[COMM]
$$\frac{j \in I \qquad \mathsf{e} \downarrow \mathsf{v}}{\mathsf{p} \triangleleft \sum_{i \in I} \mathsf{q}?\ell_i(x).P_i + \mathsf{dt}\langle a \rangle.P \;\mid\; \mathsf{q} \triangleleft \mathsf{p}!\ell_j\langle \mathsf{e} \rangle.Q \longrightarrow \mathsf{p} \triangleleft P_j\{\mathsf{v}/x\} \;\mid\; \mathsf{q} \triangleleft Q}$$

[DEFAULT]
$$\mathsf{p} \triangleleft \sum_{i \in I} \mathsf{q}?\ell_i(x).P_i + \mathsf{dt}\langle a \rangle.P \xrightarrow{\mathsf{dt}\langle a \rangle} \mathsf{p} \triangleleft P$$

[MOTION]
$$\mathsf{p} \triangleleft \mathsf{dt}\langle a \rangle.P \xrightarrow{\mathsf{dt}\langle a \rangle} \mathsf{p} \triangleleft P$$

[T-CONDITIONAL]
$$\frac{\mathsf{e} \downarrow \mathsf{true}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft P}$$

[F-CONDITIONAL]
$$\frac{\mathsf{e} \downarrow \mathsf{false}}{\mathsf{p} \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p} \triangleleft Q}$$

[R-PAR]
$$\frac{\mathsf{p} \triangleleft Q \longrightarrow \mathsf{p} \triangleleft Q'}{\mathsf{p} \triangleleft Q \mid M \longrightarrow \mathsf{p} \triangleleft Q' \mid M}$$

[M-PAR]
$$\frac{\mathsf{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle a_i \rangle} \mathsf{p}_i \triangleleft P_i' \qquad \forall i,j.\ \mathit{duration}(a_i) = \mathit{duration}(a_j)}{\Pi_i \mathsf{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle (\mathsf{p}_i : a_i) \rangle} \Pi_i \mathsf{p}_i \triangleleft P_i'}$$

[R-STRUCT]
$$\frac{M_1' \equiv M_1 \quad M_1 \longrightarrow M_2 \quad M_2 \equiv M_2'}{M_1' \longrightarrow M_2'}$$

[M-STRUCT]
$$\frac{M_1' \equiv M_1 \quad M_1 \xrightarrow{\mathsf{dt}\langle a \rangle} M_2 \quad M_2 \equiv M_2'}{M_1' \xrightarrow{\mathsf{dt}\langle a \rangle} M_2'}$$

equally for all processes; every process has to spend the same amount of time. This style of synchronisation is reminiscent of broadcast calculi [39]. Instead of broadcast messages, we broadcast *time*.

In order to state that communications can always make progress, we formalise when a multiparty session contains communications or motion actions that will never be executed.

▶ **Definition 3.2.** *A multiparty motion session M is* stuck *if $M \not\equiv \prod_{i \in I} \mathsf{p}_i \triangleleft \mathbf{0}$ and there is no multiparty session M' such that $M \longrightarrow M'$. A multiparty session M gets* stuck*, notation* $\mathtt{stuck}(M)$*, if it reduces to a stuck motion multiparty session.*

We finish this section with some examples of multi-party sessions.

▶ **Example 3.3** (A Simple Fetch Scenario)**.** Recall the scenario from Section 2 in which a cart and arm assembly has to fetch an object. There are two processes: a cart and an arm; the arm is attached to the cart. The task involves synchronization between the cart and the arm. Specifically, the protocol works as follows. Initially, the cart sends the arm a command to fold. On receiving the command, the arm folds itself. Meanwhile, the cart waits. When the arm is completely folded, it sends back a message to the cart. On receipt of this message, the cart moves, following a trajectory to the object. When it reaches the object, it stops and sends a message back to the arm to grab the object. While the cart waits, the arm executes the grabbing operation, followed by a folding operation. When the arm is done, it again synchronises with the cart. At this point, the cart moves back to its original position. (We simplify the example from Section 2 so that the sequence is not repeated.)

```
Cart◁
    Arm!fold⟨⟩.
    wait (dt⟨idle⟩){
      Arm?ok().
       dt⟨move⟩.
       Arm!grab.
       wait (dt⟨idle⟩){
         Arm?ok().
          dt⟨move⟩.
          Arm!done().0
      }
    }
```

```
Arm◁
    μX.wait (dt⟨idle⟩){
        Cart?fold().dt⟨fold⟩.Cart!ok⟨⟩.X
      + Cart?grab().
        dt⟨grip⟩.
        Cart!ok⟨⟩.X
      + Cart?done().0
    }
```

Figure 1 A cart and arm example.

Figure 1 shows how the cart and arm processes can be encoded in our core language. We introduce some syntactic sugar for readability. We write $\mathsf{wait}\ (\mathsf{dt}\langle a\rangle)\ \{\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i\}$ as shorthand for the process $\mu X.\sum_{i\in I} \mathsf{p}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.X$, which keeps running the default motion a until it receives a message.

The motion primitive idle keeps the cart or the arm stationary. The primitive move moves the cart, the primitives grip and fold respectively move the arm to grab an object or to fold the arm. At this point, we focus on the communication pattern and therefore abstract away the actual trajectories traced by the motion primitives. We come back to the trajectories in Section 5.

Finally, the multiparty session is the parallel composition of the participants Cart and Arm with the corresponding processes.

The processes in our calculus closely follow the syntax of PGCD programs [4]. In Figure 2, we show a side by side comparison of a PGCD program and the corresponding process expressed in the motion session calculus.

▶ **Example 3.4** (Multi-party Co-ordination: Handover)**.** We describe a more complex *handover* example in which a cart and arm assembly transfers an object to a second cart, called the carrier. The process for the arm is identical to Figure 1, but the cart now co-ordinates with the carrier as well. Figure 3 shows all the processes. Note that the cart now synchronises both with the arm and with the carrier.

The protocol is as follows. As before, the cart moves to a target position, having ensured that the arm is folded, and then waits for the carrier to be ready. When the carrier is ready, the arm is instructed to grab an object on the carrier. Once the object is grabbed, the arm synchronises with the cart, which then informs the carrier that the handover is complete. The cart and the carrier move back to their locations and the protocol is complete. The multiparty session is the parallel composition of the participants Cart, Arm, and Carrier, with the corresponding processes.

PGCD: pseudo code for the Arm

```
1  while true do
2  |  receive (idle)
3  |  |  fold ⇒
4  |  |  |  fold();
5  |  |  |  send(Cart, ok)
6  |  |  grab ⇒
7  |  |  |  grip();
8  |  |  |  send(Cart, ok)
9  |  |  done ⇒
10 |  |  |  break
```

Arm◁

μX.wait (dt⟨idle⟩){
  Cart?*fold*().
    dt⟨fold⟩.
    Cart!*ok*⟨⟩.X
  + Cart?*grab*().
    dt⟨grip⟩.
    Cart!*ok*⟨⟩.X
  + Cart?*done*().
    0
}

Figure 2 Comparison of a PGCD code and the corresponding motion session calculus process.

4 Multiparty Motion Session Types

This section introduces motion session types for the calculus presented in Section 3. The formulation is based on [29, 30, 14], with adaptations to account for our motion calculus.

4.1 Motion Session Types and Projections

Global types act as specifications for the message exchanges among robotic components.

▶ **Definition 4.1** (Sorts and global motion session types)**.** Sorts*, ranged over by* S*, are used to define base types:*

$$S \quad ::= \quad \texttt{unit} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{real}$$

Global types*, ranged over by* G*, are terms generated by the following grammar:*

$$G \quad ::= \quad \mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G \quad \mid \quad \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I} \quad \mid \quad \mathbf{t} \quad \mid \quad \mu\mathbf{t}.G \quad \mid \quad \texttt{end}$$

We require that $\mathsf{p} \neq \mathsf{q}$*,* $I \neq \emptyset$*,* $\ell_i \neq \ell_j$*, and* $duration(a_i) = duration(a_j)$ *whenever* $i \neq j$*, for all* $i, j \in I$*. We postulate that recursion is guarded and recursive types with the same regular tree are considered equal [37, Chapter 20, Section 2].*

In Definition 4.1, the type $\mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G$ is a *motion global type* which explicitly declares a *synchronisation* by a motion action among all the participants p_i. The rest is the standard definition of global types in multiparty session types [29, 30, 14]. The *branching* type $\mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ formalises a protocol where participant p must send to q one message with label ℓ_i and a value of type S_i as payload, for some $i \in I$; then, depending on which ℓ_i was sent by p, the protocol continues as G_i. Value types are restricted to sorts. The type `end` represents a terminated protocol. A recursive protocol is modelled as $\mu\mathbf{t}.G$, where recursion variable $\mathbf{t}$ is bound and guarded in G, e.g., $\mu\mathbf{t}.\mathbf{t}$ is not a valid type. The notation $\mathtt{pt}\{G\}$ denotes a set of participants of a global type G.

```
Cart◁
    Arm!fold⟨⟩.
    wait (dt⟨idle⟩){
      Arm?ok().Carrier!ok⟨⟩.
      dt⟨move⟩.
      wait (dt⟨idle⟩){
        Carrier?ok().Arm!grab⟨⟩.
        wait (dt⟨idle⟩){
          Arm?ok().Carrier!ok⟨⟩.
          dt⟨move⟩.
          Arm!done⟨⟩.Carrier!done⟨⟩.0
        }
      }
    }
```

```
Carrier◁
    wait (dt⟨idle⟩){
      Cart?ok().dt⟨move⟩.
        Cart!ok⟨⟩.
        wait (dt⟨idle⟩){
          Cart?ok().
            dt⟨move⟩.
            wait (dt⟨idle⟩){Cart?done().0}
    }}
Arm◁
    μX.wait (dt⟨idle⟩){
         Cart?fold().dt⟨fold⟩.Cart!ok⟨⟩.X
      + Cart?grab().dt⟨grip⟩.Cart!ok⟨⟩.X
      + Cart?done().0
    }
```

Figure 3 A multi-party handover example.

▶ **Example 4.2** (Global session types)**.** The global session type for the fetch example (Example 3.3) is:

$$
\begin{aligned}
&\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.\\
&\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
&\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.\\
&\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
&\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathtt{end}
\end{aligned}
$$

and the global session type for the handover example (Example 3.4) is:

$$
\begin{aligned}
&\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.\\
&\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).\\
&\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
&\mathsf{Carrier} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).\\
&\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.\\
&\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).\\
&\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.\\
&\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{done}(\mathtt{unit}).\mathtt{end}
\end{aligned}
$$

A *(local) motion session type* describes the behaviour of a single participant in a multiparty motion session.

▶ **Definition 4.3** (Local motion session types). *The grammar of local types, ranged over by T, is:*

$$T \quad ::= \quad \mathsf{dt}\langle a\rangle.T \quad | \quad \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \quad | \quad \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \,\&\, \mathsf{dt}\langle a\rangle.T \quad | \quad \oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$$
$$| \quad \mathbf{t} \quad | \quad \mu\mathbf{t}.T \quad | \quad \mathtt{end}$$

We require that $\ell_i \neq \ell_j$ whenever $i \neq j$, for all $i, j \in I$. We postulate that recursion is always guarded. Unless otherwise noted, session types are closed.

Labels in a type need to be pairwise different, e.g., $\mathsf{p}?\ell(\mathtt{int}).\mathtt{end}\&\mathsf{p}?\ell(\mathtt{nat}).\mathtt{end}$ is not a type. The *motion local type* $\mathsf{dt}\langle a\rangle.T$ represents a motion action followed by the type T; the *external choice* or *branching type* $\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I}$ requires to wait to receive a value of sort S_i (for some $i \in I$) from the participant p, via a message with label ℓ_i; if the received message has label ℓ_i, the protocol will continue as prescribed by T_i. The *motion branching choice* is equipped with a default motion type $\mathsf{dt}\langle a\rangle.T$. The *internal choice* or *selection type* $\oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$ says that the participant implementing the type must choose a labelled message to send to q; if the participant chooses the message ℓ_i, for some $i \in I$, it must include in the message to q a payload value of sort S_i, and continue as prescribed by T_i. Recursion is modelled by the session type $\mu\mathbf{t}.T$. The session type $\mathtt{end}$ says that no further communication is possible and the protocol is completed. We adopt the following conventions: we do not write branch/selection symbols in case of a singleton choice, we do not write unnecessary parentheses, and we often omit trailing $\mathtt{end}$s. The notation $\mathtt{pt}\{T\}$ denotes a set of participants of a session type T.

In Definition 4.4 below, we define the *global type projection* as a relation $G \upharpoonright_{\mathsf{r}} T$ between global and local types. Our definition extends the one originally proposed by [25, 26], along the lines of [12] and [13] with motion types: i.e., it uses a *merging operator* $\bigsqcap$ to combine multiple session types into a single type.

▶ **Definition 4.4.** *The* projection of a global type onto a participant r *is the largest relation $\upharpoonright_r$ between global and session types such that, whenever $G \upharpoonright_r T$:*

- $G = \mathtt{end}$ *implies* $T = \mathtt{end}$; [PROJ-END]
- $G = \mathsf{dt}\langle(\mathsf{p}_i : a_i)\rangle.G'$ *implies* $T = \mathsf{dt}\langle a_j\rangle.T'$ *with* $r = \mathsf{p}_j$ *and* $G' \upharpoonright_r T'$; [PROJ-MOTION]
- $G = \mathsf{p} \to \mathsf{r} : \{\ell_i(S_i).G_i\}_{i\in I}$ *implies* $T = \&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I}$ *with* $G_i \upharpoonright_r T_i$; [PROJ-IN]
- $G = \mathsf{r} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ *implies* $T = \oplus\{\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$ *and* $G_i \upharpoonright_r T_i$, $\forall i \in I$; [PROJ-OUT]
- $G = \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).G_i\}_{i\in I}$ *and* $\mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\}$ *implies that there are* T_i, $i \in I$ *s.t.* $T = \bigsqcap_{i\in I} T_i$, *and* $G_i \upharpoonright_r T_i$, *for every* $i \in I$. [PROJ-CONT]
- $G = \mu\mathbf{t}.\mathsf{G}$ *implies* $T = \mu\mathbf{t}.T'$ *with* $G \upharpoonright_r T'$ *if* r *occurs in* G, *otherwise* $T = \mathtt{end}$. [PROJ-REC]

Above, $\bigsqcap$ is the merging operator*, that is a partial operation over session types defined as:*

$$
T_1 \bigsqcap T_2 \;=\; \begin{cases}
T_1 & \text{if } T_1 = T_2 & \text{[MRG-ID]}\\
T_3 & \text{if } \exists I,J: \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} & \text{and}\\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J} & \text{and}\\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J} \end{cases} & \text{[MRG-BRA1]}\\
T_3 & \text{if } \exists I,J: \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I}\&\ \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J}\&\ \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J}\&\ \mathsf{dt}\langle a\rangle.T' \end{cases} & \text{[MRG-BRA2]}\\
T_3 & \text{if } \exists I,J: \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I} & \text{and}\\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{j\in J}\&\ \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_3 = \&\{\mathsf{p}'?\ell_k(S_k).T_k\}_{k\in I\cup J}\&\ \mathsf{dt}\langle a\rangle.T' \end{cases} & \text{[MRG-BRA3]}\\
T_3 & \text{if } \exists I,J: \begin{cases} T_1 = \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_2 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I}\&\ \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_3 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I}\&\ \mathsf{dt}\langle a\rangle.T' \end{cases} & \text{[MRG-BRA4]}\\
T_3 & \text{if } \exists I,J: \begin{cases} T_1 = \mathsf{dt}\langle a\rangle.T' & \text{and}\\ T_2 = \&\{\mathsf{p}'?\ell_j(S_j).T_j\}_{i\in I} & \text{and}\\ T_3 = \&\{\mathsf{p}'?\ell_i(S_i).T_i\}_{i\in I}\&\ \mathsf{dt}\langle a\rangle.T' \end{cases} & \text{[MRG-BRA5]}\\
T_2 \bigsqcap T_1 & \text{if } T_2 \bigsqcap T_1 \text{ is defined,} & \\
\textit{undefined} & \text{otherwise.} &
\end{cases}
$$

We omit the cases for recursions and selections (defined as [42, S 3]).

Note that our definition is slightly simplified w.r.t. the one of [12] and [13]. Instead of this mergeability operator, one might use more general approach from [42]. This definition is sufficient for our purposes (i.e., to demonstrate an application of session types to robotics communications).

▶ **Example 4.5.** The projection of the global session type for the fetch example on the cart gives the following local session type:

$$\mathsf{Arm}!\mathit{fold}\langle\mathtt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Arm}?\mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle\mathsf{move}\rangle.\mathsf{Arm}!\mathit{grab}\langle\mathtt{unit}\rangle.\\ \mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Arm}?\mathit{ok}(\mathtt{unit}).\mathsf{dt}\langle\mathsf{move}\rangle.\mathsf{Arm}!\mathit{done}\langle\mathtt{unit}\rangle.\mathtt{end}$$

The local motion session type for the arm is:

$$\mathsf{Cart}?\mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle\mathsf{fold}\rangle.\mathsf{Cart}!\mathit{ok}\langle\mathtt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Cart}?\mathit{grab}(\mathtt{unit}).\\ \mathsf{dt}\langle\mathsf{grip}\rangle.\mathsf{Cart}!\mathit{ok}\langle\mathtt{unit}\rangle.\mathsf{dt}\langle\mathsf{idle}\rangle.\mathsf{Cart}?\mathit{done}(\mathtt{unit}).\mathtt{end}$$

On Progress of Time. Our model assumes that the computation and message transmission time is much faster than the dynamics of the system and, therefore, the messages can be seen as instantaneous. This assumption depends on parameters of the system, like the speed of the network and the dynamics of the physical system, and also on the program being executed. While we cannot directly change the physical system, we can at least check the program is well behaved w.r.t. to time.

If a program can send an unbounded number of messages without executing a motion then this assumption, obviously, does not hold. From the perspective of using the motion calculus to verify a system, this may lead to situation where an unsafe program is deemed safe because time does not progress. For instance, a robot driving straight into a wall could "avoid" crashing into the wall by sending messages in a loop and, therefore, stopping the progress of time.

This problem is not unique to our system but a more general problem in defining the semantics of hybrid systems [20, 21]. In general, one needs to assume that time always *diverges* for infinite executions. In this work, we take a pragmatic solution and simply *disallow*

0-time recursion. When recursion is used, all the paths between a $\mu \mathbf{t}$ and the corresponding $\mathbf{t}$ must contain at least one motion primitive. This is a simple check which can be done at the syntactic level of global types and it is a sufficient condition for forcing the progress of time.

4.2 Motion Session Typing

We now introduce a type system for the multiparty session calculus presented in Section 3. We distinguish three kinds of typing judgments:

$$\Gamma \vdash \mathsf{e} : S \qquad \Gamma \vdash P : T \qquad \vdash M : G$$

where Γ is the *typing environment* defined as: $\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, X : T$, i.e., a mapping that associates expression variables with sorts, and process variables with session types.

We use the subtyping relation $\leqslant$ to augment the flexibility of the type system by determining when a type T is "smaller" than T', it allows to use a process typed by the former whenever a process typed by the latter is required.

▶ **Definition 4.6** (Subsorting and subtyping)**.** *Subsorting* $\leq:$ *is the least reflexive binary relation such that* $\mathtt{nat} \leq: \mathtt{int} \leq: \mathtt{real}$*. Subtyping* $\leqslant$ *is the largest relation between session types coinductively defined by the following rules:*

$$[\text{SUB-END}]\quad \mathtt{end} \leqslant \mathtt{end} \qquad [\text{SUB-IN1}]\quad \frac{\forall i \in I : \quad S_i' \leq: S_i \quad T_i \leqslant T_i' \quad T \leqslant T'}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I\cup J} \,\&\, \mathsf{dt}\langle a\rangle.T \leqslant \&\{\mathsf{p}?\ell_i(S_i').T_i'\}_{i\in I} \,\&\, \mathsf{dt}\langle a\rangle.T'}$$

$$[\text{SUB-MOTION}]\quad \frac{T \leqslant T'}{\mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a\rangle.T'} \qquad [\text{SUB-IN2}]\quad \frac{\forall i \in I : \quad S_i' \leq: S_i \quad T_i \leqslant T_i'}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I\cup J} \,\&\, \mathsf{dt}\langle a\rangle.T \leqslant \&\{\mathsf{p}?\ell_i(S_i').T_i'\}_{i\in I}}$$

$$[\text{SUB-IN3}]\quad \frac{T \leqslant T'}{\&\{\mathsf{p}?\ell_i(S_i).T_i\}_{i\in I} \,\&\, \mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a\rangle.T'} \qquad [\text{SUB-OUT}]\quad \frac{\forall i \in I : \quad S_i \leq: S_i' \quad T_i \leqslant T_i'}{\oplus\{\mathsf{p}!\ell_i(S_i).T_i\}_{i\in I} \leq: \oplus\{\mathsf{p}!\ell_i(S_i').T_i'\}_{i\in I\cup J}}$$

The double line in the subtyping rules indicates that the rules are interpreted *coinductively* [37, Chapter 21].

The typing rules for expressions are given as expected and omitted. The typing rules for processes and multiparty sessions are the content of Table 2:

- [T-SUB] is the *subsumption rule*: a process with type T is also typed by the supertype T';
- [T-**0**] says that a terminated process implements the terminated session type;
- [T-REC] types a recursive process $\mu X.P$ with T if P can be typed as T, too, by extending the typing environment with the assumption that X has type T;
- [T-VAR] uses the typing environment assumption that process X has type T;
- [T-MOTION] types a motion process as a motion local type;
- [T-INPUT-CHOICE] types a summation of input prefixes as a branching type and a default branch as a motion type. It requires that each input prefix targets the same participant q, and that, for all $i \in I$, each continuation process P_i is typed by the continuation type T_i, having the bound variable x_i in the typing environment with sort S_i. Note that the rule implicitly requires the process labels ℓ_i to be pairwise distinct (as per Definition 4.3);

Table 2 Typing rules for motion processes.

$$[\text{T-}\mathbf{0}] \quad \Gamma \vdash \mathbf{0} : \mathsf{end} \qquad [\text{T-REC}] \quad \frac{\Gamma, X : T \vdash P : T}{\Gamma \vdash \mu X.P : T} \qquad [\text{T-VAR}] \quad \Gamma, X : T \vdash X : T \qquad [\text{T-MOTION}] \quad \frac{\Gamma \vdash Q : T}{\Gamma \vdash \mathsf{dt}\langle a\rangle.Q : \mathsf{dt}\langle a\rangle.T}$$

$$[\text{T-OUT}] \quad \frac{\Gamma \vdash \mathsf{e} : S \quad \Gamma \vdash P : T}{\Gamma \vdash \mathsf{q}!\ell(\mathsf{e}).P : \mathsf{q}!\ell(S).T} \qquad [\text{T-INPUT-CHOICE1}] \quad \frac{\forall i \in I \quad \Gamma, x_i : S_i \vdash P_i : T_i}{\Gamma \vdash \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i : \&\{\mathsf{q}?\ell_i(S_i).T_i\}_{i\in I}}$$

$$[\text{T-INPUT-CHOICE2}] \quad \frac{\forall i \in I \qquad \Gamma, x_i : S_i \vdash P_i : T_i \qquad \Gamma \vdash \mathsf{dt}\langle a\rangle.Q : T}{\Gamma \vdash \sum_{i\in I} \mathsf{q}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.Q : \&\{\mathsf{q}?\ell_i(S_i).T_i\}_{i\in I} \,\&\, T}$$

$$[\text{T-CHOICE}] \quad \frac{\Gamma \vdash \mathsf{e} : \mathtt{bool} \qquad \exists k \in I \qquad \Gamma \vdash P_1 : T_k \qquad \Gamma \vdash P_2 : \oplus\{T_i\}_{i\in I\setminus\{k\}}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{T_i\}_{i\in I}}$$

$$[\text{T-SUB}] \quad \frac{\Gamma \vdash P : T \quad T \leqslant T'}{\Gamma \vdash P : T'} \qquad [\text{T-SESS}] \quad \frac{\forall i \in I \quad \vdash P_i : G\upharpoonright \mathsf{p}_i \qquad \mathtt{pt}\{G\} = \{\mathsf{p}_i \mid i \in I\}}{\vdash \prod_{i\in I} \mathsf{p}_i \triangleleft P_i : G}$$

- [T-OUT] types an output prefix with a singleton selection type, provided that the expression in the message payload has the correct sort S, and the process continuation matches the type continuation;
- [T-CHOICE] types a conditional process by matching the branches of the types to branches of the sub-processes;
- [T-SESS] types multiparty sessions, by associating typed processes to participants. It requires that the processes being composed in parallel can play as participants of a global communication protocol: hence, their types must be projections of a single global type G. As the temporal evolution (motion) synchronises all the processes condition $\mathtt{pt}\{G\} = \{\mathsf{p}_i \mid i \in I\}$ guarantees that motions are defined for every participant.

▶ **Example 4.7.** We sketch the main steps to show that the Arm process is typed by the local type from Example 4.5. The type derivation uses the subtyping rules. This is because the process for the arm makes an external choice between the messages *fold*, *grab*, *done*, and the default motion primitive idle, and the type fixes a specific sequence of messages. The usual subtyping rules [SUB-IN1] and [SUB-IN2] allow typing the process against the local type, by "expanding" the local type with the other possible choices. The interesting subtyping rule is [SUB-IN3], which states that an external choice with a default motion type refines only the default motion type. This is needed to type the process against the local type

$$\mathsf{dt}\langle \mathsf{idle}\rangle.\mathsf{Cart}?\mathit{grab}(\mathtt{unit}).T$$

This subtyping rule is sound, because the local type ensures that the other message choices cannot arise.

The proposed motion session type system satisfies two fundamental properties: typed sessions only reduce to typed sessions (subject reduction), and typed sessions never get stuck.

In order to state subject reduction, we need to formalise how global types are reduced when local session types reduce and evolve. Note that since the same motion actions always synchronise among all participants, they always make progress (hence they are always consumed).

▶ **Definition 4.8** (Global types consumption and reduction). *The* consumption *of the communication* $p \xrightarrow{\ell} q$ *and motion* $\mathsf{dt}\langle a\rangle$ *for the global type* G *(notation* $G \setminus p \xrightarrow{\ell} q$ *and* $G \setminus \mathsf{dt}\langle a\rangle$*) is the global type defined (up to unfolding of recursive types) as follows:*

$$\mathsf{dt}\langle a\rangle.G \setminus \mathsf{dt}\langle a\rangle \;=\; G$$

$$\big(p \to q : \{\ell_i(S_i).G_i\}_{i\in I}\big) \setminus p \xrightarrow{\ell} q \;=\; G_k \qquad \text{if } \exists k \in I : \ell = \ell_k$$

$$\big(r \to s : \{\ell_i(S_i).G_i\}_{i\in I}\big) \setminus p \xrightarrow{\ell} q \;=\; r \to s : \{\ell_i(S_i).G_i \setminus p \xrightarrow{\ell} q\}_{i\in I} \quad \text{if } \{r,s\} \cap \{p,q\} = \emptyset \;\wedge\; \forall i \in I : \{p,q\} \subseteq G_i$$

The reduction of global types *is the smallest pre-order relation closed under the rule:* $G \Longrightarrow G \setminus p \xrightarrow{\ell} q$ *and* $G \Longrightarrow G \setminus \mathsf{dt}\langle a\rangle$.

We can now state the main results.

▶ **Theorem 4.9** (Subject Reduction). *Let* $\vdash M : G$*. For all* M'*, if* $M \longrightarrow M'$*, then* $\vdash M' : G'$ *for some* G' *such that* $G \Longrightarrow G'$.

▶ **Corollary 4.10.** *Let* $\vdash M : G$*. If* $M \longrightarrow^* M'$*, then* $\vdash M' : G'$ *for some* G' *such that* $G \Longrightarrow G'$.

▶ **Theorem 4.11** (Progress). *If* $\vdash M : G$*, then either* $M \equiv \prod_{i\in I} p_i \triangleleft \mathbf{0}$ *or there is* M' *such that* $M \longrightarrow M'$.

As a consequence of subject reduction and progress, we get the safety property stating that a typed multiparty session will never get stuck.

▶ **Theorem 4.12** (Type Safety). *If* $\vdash M : G$*, then it does not hold* $\mathtt{stuck}(M)$.

Proof. Direct consequence of Corollary 4.10, Theorem 4.11, and Definition 3.2. ◀

5 Motion Primitives: Trajectories and Resources

So far, our motion calculus abstracted the trajectories of the robots and only considered the time it takes to execute motion primitives. This is sufficient to show that the synchronisation and communication protocol between the robots executes correctly. However, it is too abstract to prove more complex properties about executions of the system. In particular, for an execution to proceed correctly we need to check the existence of trajectories for all the robots. A joint trajectory may not exist, for example, if the motion primitives cause a collision in the physical world.

In this section, we explain how to make our model more detailed and how to look inside the motion primitives for the continuous evolution of trajectories. To accomplish this, first, we give a semantics that includes trajectories. Then, we refine our calculus to replace internal choice with guarded choice. Finally, we explain how to use session types to prove properties over the trajectories.

5.1 Model for the Robots and Motion Primitives

We proceed following the formalisation of trajectories in the PGCD language for robotics [4].

Robots. Each participant $(\mathsf{p}, \mathsf{q}, \ldots)$ maintains a state in the physical world. This state is updated when its own motion primitives execute as well as on potential physical interactions with other processes.

We model the physical state of a process as a tuple $(\mathit{Var}, \rho, \mathit{rsrc})$ where Var is a set of variables, with two distinguished disjoint subsets X and W of *physical state* and *external input* variables, $\rho : \mathit{Var} \to \mathbb{R}$ is a *store* mapping variables to values, and rsrc is a *resource function* mapping a store to a subset of $\mathbb{R}^3$. The resource function represents the geometric footprint in space occupied by the robot. We shall use this function to check the absence of collisions between robots.

When two robots p_1 and p_2 are in the same environment, we may connect some state variables of one process to the external inputs of the other. This represents physical coupling between these robots. A *connection* θ 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 $x \in \mathsf{p}_1.X$ and $w \in \mathsf{p}_2.W$ or $x \in \mathsf{p}_2.X$ and $w \in \mathsf{p}_1.W$, and (2) there does not exist $(x, w), (x', w) \in \theta$ such that x and x' are distinct. Two connections θ_1 and θ_2 are *compatible* if $\theta_1 \cup \theta_2$ is a connection. We assume that all the participants in a session are connected by compatible connections.

For example, consider a cart and an arm. The physical variables can provide the position and velocities of the center of mass of the cart and of the arm. Note that if the arm is attached to the cart, then its position changes when the cart moves. Thus, the position and velocity of the cart are external inputs to the arm, and play a role in determining its own position. However, the arm can also move relative to the cart and the position of its end effector is determined both by the external inputs as well as its relative position and velocity. Furthermore, the mass and the position of the center of mass of the arm are external inputs to the cart, because these variables affect the dynamics of the cart.

Motion Primitives. Let X and W be two sets of real-valued variables, representing internal state and external input variables of a robotic system, respectively. A motion primitive updates the values of the variables in X 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 (ξ, ν), i.e., a valuation over time to variables in X and W.

Formally, a motion primitive m is a tuple $(T, \mathrm{Pre}, \mathrm{Inv}, \mathrm{Post})$ consisting of a *duration* T, a *pre-condition* $\mathrm{Pre} \subseteq \mathbb{R}^{|X|} \times \mathbb{R}^{|W|}$, an *invariant* $\mathrm{Inv} \subseteq \big([0,T] \to \mathbb{R}^{|X|}\big) \times \big([0,T] \to \mathbb{R}^{|W|}\big)$, and a *post-condition* $\mathrm{Post} \subseteq \mathbb{R}^{|X|} \times \mathbb{R}^{|W|}$. A *trajectory* of duration T of the motion primitive m is a pair of continuous functions (ξ, ν) mapping the real interval $[0, T]$ to $\mathbb{R}^{|X|}$ and $\mathbb{R}^{|W|}$, respectively, such that $(\xi, \nu) \in \mathrm{Inv}$, $(\xi(0), \nu(0)) \in \mathrm{Pre}$, and $(\xi(T), \nu(T)) \in \mathrm{Post}$.

Correspondingly, we need to update the semantics of our motion calculus:

- The participant executing a program $\mathsf{p} \triangleleft P$ now also carries a store containing a valuation for the physical state of the robot: $\mathsf{p}, \rho \triangleleft P$.
- The motion transitions $\xrightarrow{\mathsf{dt}\langle a \rangle}$ get labelled with trajectories: $\xrightarrow{\mathsf{dt}\langle (\xi,\nu) \rangle}$.
- The semantics rule for choice can use values from the store:

$$\frac{\rho(\mathsf{e}) \downarrow \mathsf{true}}{\mathsf{p}, \rho \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p}, \rho \triangleleft P}\ [\textsc{t-conditional}] \qquad \frac{\rho(\mathsf{e}) \downarrow \mathsf{false}}{\mathsf{p}, \rho \triangleleft \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow \mathsf{p}, \rho \triangleleft Q}\ [\textsc{f-conditional}]$$

where $\rho(\mathsf{e})$ replaces the variables from Var in e with their value according to ρ.

- The semantics of a motion checks the trajectories against the motion primitive specification and the store:

$$\text{[MOTION]}\quad \frac{\begin{array}{c} a = (T, \mathrm{Pre}, \mathrm{Inv}, \mathrm{Post}) \qquad \mathit{range}(\xi) = [0,T] \qquad \rho = \xi(0) \qquad \rho' = \xi(T) \\ (\xi(0), \nu(0)) \in \mathrm{Pre} \qquad (\xi(T), \nu(T)) \in \mathrm{Post} \qquad \forall t \in [0,T].\, (\xi(t), \nu(t)) \in \mathrm{Inv} \end{array}}{\mathsf{p}, \rho \triangleleft \mathsf{dt}\langle a \rangle.P \xrightarrow{\mathsf{dt}\langle(\xi,\nu)\rangle} \mathsf{p}, \rho' \triangleleft P}$$

  The rule checks that the trajectory is valid w.r.t. a: the duration of the trajectory must match the duration of the motion primitive, the start and end of the trajectory match the state of ρ and ρ' respectively. Furthermore, the pre-condition, post-condition, and invariant must be respected.
- The parallel composition of motions connects the external inputs of each process according to the connections. For the notations, we use subscript to denote that an element belongs to a particular process p, e.g., X_{p} for the internal variables of p. We denote the restriction of a trajectory ξ over a subset X of the dimensions by $\xi|_X$.

$$\text{[M-PAR]}\quad \frac{\begin{array}{c} \forall i \qquad \xi_i = \xi|_{X_{\mathsf{p}_i}} \qquad \nu_i = \theta_{\mathsf{p}_i}(\xi)|_{W_{\mathsf{p}_i}} \qquad \mathsf{p}_i, \rho_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle(\xi_i,\nu_i)\rangle} \mathsf{p}_i, \rho_i' \triangleleft P_i' \\ \forall i,j,t.\ i \neq j \Rightarrow \mathit{rsrc}_{\mathsf{p}_i}(\xi|_{X_{\mathsf{p}_i}}(t), \theta_{\mathsf{p}_i}(\xi)|_{W_{\mathsf{p}_i}}(t)) \cap \mathit{rsrc}_{\mathsf{p}_j}(\xi|_{X_{\mathsf{p}_j}}(t), \theta_{\mathsf{p}_j}(\xi)|_{W_{\mathsf{p}_j}}(t)) = \emptyset \end{array}}{\Pi_i \mathsf{p}_i, \rho_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle(\xi,\nu)\rangle} \Pi_i \mathsf{p}_i, \rho_i' \triangleleft P_i'}$$

  Even at the top level, there is a ν as there can be elements which are under the control of the environment. Then, for each process we create the appropriate trajectory (ξ, ν) by applying the appropriate connection θ. Also, the resources used by each participants during the motion needs to disjoint from each other. This last check ensures the absence of collision between robots. We use this check to avoid the complexity of modelling collisions.

▶ **Example 5.1.** Let us look at the cart from Example 3.3. The cart is moving on the ground, a 2D plane and, therefore, we model its physical state (X_{Cart}) by its position $\mathbf{p}_{\mathsf{Cart}} \in \mathbb{R}^2$, orientation $\mathbf{r}_{\mathsf{Cart}} \in [-\pi; \pi)$, and speed $\mathbf{s}_{\mathsf{Cart}} \in \mathbb{R}$.

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 $\mathbf{s}_{\mathsf{Cart}} = 0$ (i.e., it is at rest), the post-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_0 \wedge \mathbf{r}_{\mathsf{Cart}} = \mathbf{r}_0$, and the invariant is $\mathbf{p}_{\mathsf{Cart}}(t) = \mathbf{p}_0 \wedge \mathbf{r}_{\mathsf{Cart}}(t) = \mathbf{r}_0 \wedge \mathbf{s}_{\mathsf{Cart}} = 0$ for all $t \in [0, T]$.

A slightly more interesting motion primitive is $\mathsf{move}(\mathbf{p}_0, \mathbf{p}_t)$, which moves the cart from position $\mathbf{p}_0$ to $\mathbf{p}_t$. The pre-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_0$. The post-condition is $\mathbf{s}_{\mathsf{Cart}} = 0 \wedge \mathbf{p}_{\mathsf{Cart}} = \mathbf{p}_t$. The invariant can specify a bound on the velocity, e.g., $0 \leq \mathbf{s}_{\mathsf{Cart}} \leq v_{\max}$, that the cart moves in straight line between $\mathbf{p}_0$ and $\mathbf{p}_t$, etc.

We can also include external input. For instance, we may add an external variable $\mathbf{w_{obj}}$ to represent the weight of any carried object, e.g., the arm attached on top. Then, the pre-condition of move may include an extra constraint $0 \leq \mathbf{w_{obj}} \leq w_{\max}$ to say that the cart can only move if the weight of the payload is smaller than a given bound.

5.2 Motion Calculus with Guarded Choice

Before executing some motion, a process may need to test the state of the physical world and, according to the current state, decide what to do. Therefore, we extend the calculus with the ability for a process to test predicates over its Var as part of the $\mathsf{if}\ \cdot\ \mathsf{then}\ \cdot\ \mathsf{else}\ \cdot$. On the specification side, we also add predicates to the internal choice.

Let $\mathcal{P}$ range over predicates. The global and local motion session types are modified as follows:

- The branching type for global session types becomes $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$.
- The branching type for local session types becomes $\oplus\{[\mathcal{P}_i]\mathsf{q}!\ell_i(S_i).T_i\}_{i\in I}$.

To make sure the modified types can be projected and then used for typing they need to respect the following constraints. Assume that Var_{p} are the variables associated with the robot executing the role of p. (1) The choices are *local*, i.e., for $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$ we have that $\mathrm{fv}(\mathcal{P}_i) \subseteq Var_{\mathsf{p}}$ for all i in I. (2) The choices are *total*, i.e., for $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(S_i).G_i\}_{i\in I}$ we have that $\bigvee_{i\in I} \mathcal{P}_i$ is valid. The local types have similar constraints.

The subtyping and typing relation are updated as follows:

[SUB-OUT]
$$\frac{\forall i \in I : \quad S_i \leq: S_i' \quad T_i \leqslant T_i' \quad \mathcal{P}_i \Rightarrow \mathcal{P}_i'}{\oplus\{[\mathcal{P}_i]\mathsf{p}!\ell_i(S_i).T_i\}_{i\in I} \leqslant \oplus\{[\mathcal{P}_i']\mathsf{p}!\ell_i(S_i').T_i'\}_{i\in I\cup J}}$$

The change in this rule is the addition of checking the implication $\mathcal{P}_i \Rightarrow \mathcal{P}_i'$ to make sure that if the pre-condition of a motion primitive relies on $\mathcal{P}_i'$, it still holds with $\mathcal{P}_i$. Notice that $\oplus\{[\mathcal{P}_i]\mathsf{p}!\ell_i(S_i).T_i\}_{i\in I}$ which can have more restricted predicates needs to be a valid local type and the guards still need to be total.

[T-CHOICE]
$$\frac{\Gamma \vdash \mathsf{e} : \mathtt{bool} \quad \exists k \in I \quad \mathsf{e} \Rightarrow \mathcal{P}_k \quad \Gamma \vdash P_1 : T_k \quad \Gamma \vdash P_2 : \oplus\{[e \vee \mathcal{P}_i]T_i\}_{i\in I\setminus\{k\}}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{[\mathcal{P}_i]T_i\}_{i\in I}}$$

Type checking the rules propagates the expression from if then else and matches it into a branch of the type. To deal with the else branch we modify the predicate in the remaining branches of the type. For the last else branch of a, possibly nested, if then else we need the following extra rule:

[T-CHOICE-FINAL]
$$\frac{\Gamma \vdash P : T}{\Gamma \vdash P : \oplus\{[\mathsf{true}]T\}}$$

▶ **Example 5.2.** Usually, for the propagation of tested expressions through the branches we modify the type. Let us make an example of how this works. Consider we have the following process $\mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ which has the type $\oplus\{[\mathsf{e}_1]T_1, [\neg\mathsf{e}_1]T_2\}$. Assuming that $P_i : T_i$ for $i \in \{1, 2\}$ we can build the following derivation:

$$\frac{\mathsf{e}_1 \Rightarrow \mathsf{e}_1 \qquad \Gamma \vdash P_1 : T_1 \qquad \dfrac{\Gamma \vdash P_2 : T_2}{\Gamma \vdash P_2 : \oplus\{[\mathsf{e}_1 \vee \neg\mathsf{e}_1]T_2\}}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}_1\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{[\mathsf{e}_1]T_1, [\neg\mathsf{e}_1]T_2\}}$$

With a bit of boolean algebra, we can show that $\mathsf{e}_1 \vee \neg\mathsf{e}_1 \Leftrightarrow \mathsf{true}$.

5.3 Existence of Joint Trajectories and Verification

The goal of the compatibility check is to make sure that abstract motion primitives specified in a global type can execute concurrently. This requires two checks. First, for motion primitives of different processes executed in parallel, we need to make sure that there exists a trajectory

satisfying all the constraints of the motion primitives. Second, for motion primitives executed sequentially by the same process, we need to make sure that the post-condition of the first implies the pre-condition of the second motion primitive, taking into account the guards of choices in the middle.

To check that motion primitives executing in parallel have a joint trajectory, we use an assume-guarantee style of reasoning. 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. We refer to standard methods [35, 4] for the details.

For the allowed trajectories, we need to also check the absence of collision. This means that once we have the constrains defining a joint trajectory ξ to check that for any two distinct processes p and q the property $rsrc_{\mathsf{p}}(\xi_{\mathsf{p}}) \cap rsrc_{\mathsf{q}}(\xi_{\mathsf{q}}) = \emptyset$.

▶ **Example 5.3.** In Example 3.4, the cart and the carrier are moving toward each other. They need to be close enough for the arm to grab the object but far enough to avoid colliding. We model the resources of the cart by a cylinder around the cart's position: $rsrc_{\mathsf{Cart}} = \{(x, y, z) || (x, y) - \mathbf{p}_{\mathsf{Cart}}| \leq r \wedge 0 \leq z \leq h\}$ where r is the "radius" of the cart and h its height. The carrier's resources are similar but with the appropriate radius and height r', h'. The cart and carrier does not collide if we can prove that $\forall t. \mid \xi_{\mathsf{Cart}}|_{\mathbf{p}_{\mathsf{Cart}}}(t) - \xi_{\mathsf{Carrier}}|_{\mathbf{p}_{\mathsf{Carrier}}}(t) \mid > r + r'$.

6 Evaluation

6.1 Implementation

We have implemented the system we describe on top of PGCD [4][1], a system for programming and verification of robotic systems. PGCD is build on top of the Robotic Operating System (ROS) [40], a software ecosystem for robots. The core of ROS is a publish-subscribe messaging system. PGCD uses ROS's messaging to implement its synchronous message-passing layer. On the verification side, PGCD uses a mix of model-checking (using Spin [22]) to deal with the message-passing structure, and symbolic reasoning (using SymPy [33]) and constraint solving (using dReal [16]) to reason about motion primitives.

We replace the global model-checking algorithm of PGCD with motion session calculus specifications but reuse PGCD's infrastructure to reason about the trajectories of motion primitives. Currently, our implementation uses a syntax for specifications closer to the state-machine form of session types [11] but without the parallel composition operator. This representation allows for more general guarded choice. $\mathsf{if} \cdot \mathsf{then} \cdot \mathsf{else} \cdot$ implicitly forces disjoint guards for the two branches. Our implementation allows overlapping guards. Algorithmically, since the types are represented in a form close to an automaton, the projection and merge operations are implemented using automata theoretic operation: morphism, minimisation, and checking determinism of the result.

The typing, including subtyping, is implemented by computing an alternating simulation [2] between programs and their respective local type. Intuitively, an alternative refinement relation check that a process implements its specification (subset of the behaviours) without restricting the other processes. For synchronous message passing programs, the subtyping relation for session type matches alternating refinement. We use this view on subtyping as the theory of alternating simulation [2] gives us an algorithm to compute this relation and, therefore, check subtyping.

[1] PGCD repository is `https://github.com/MPI-SWS/pgcd`. The code for this work is located in the `pgcd/nodes/verification/choreography` folder.

6.2 Experiments

For the evaluation, we take two existing PGCD programs and write global types in motion session calculus that describe the co-ordination in the program.

First, we describe our experimental setup, both for the hardware and for the software. Then, we explain the experiments. Finally, we report on the size of the specifications, and time to check the programs satisfy the specification.

Setup

We use three robots: a robotic arm and two carts, shown in Figure 4. The robots are built with a mix of off-the-self parts and 3D printed parts.

Arm. The arm is a modified BCN3D MOVEO,[2] where the upper arm section is shortened to make it lighter and easier to mount on the cart. The arm with its control electronics is mounted on top of the cart.

Cart. The cart is shown on Figure 4a. The control electronics and motors are situated below the wooden board. The cart is an omnidirectional driving platform. It uses omniwheels to get three degrees of freedom (two in translation, one in rotation) and can move between any two positions 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. Due to the large power consumption of the arm mounted on top, this cart is powered by a tether.

Carrier. We call the second cart the carrier (Figure 4b) as we use it to carry the block that is grabbed by the arm. As the first cart, it is also omnidirectional (mecanum wheels).

All the three robots use stepper motors to move precisely. The robots 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 turns. 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 known accurately. In our experiments, we use markings on the ground to fix the initial state as can be seen in Figure 5. 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 to run the program. The ROS master node, providing core messaging services, runs on a separate laptop to which all the robots connect. The RaspberryPi runs Raspbian OS (based on Debian Jessie) and the laptop runs Ubuntu 16.04. The ROS version is Kinetic Kame.

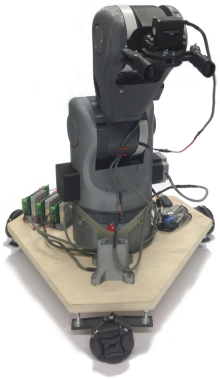
Experiments

We describe two experiments:

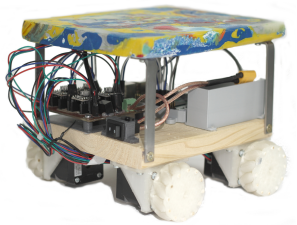
Handover. This experiment corresponds to our earlier example. The two carts meet before the arm takes an object placed on top of the carrier and, then, they go back to their initial position (see Figure 5a).

Underpass. First, the carrier cart brings an object to the arm which is then taken by the arm. Then, the carrier cart goes around the arm passing under an obstacle which is high enough for just the carrier alone. Finally, the arm puts the object back on the carrier on the other side of the obstacle. This can be seen in Figure 5b.

[2] `https://github.com/BCN3D/BCN3D-Moveo`

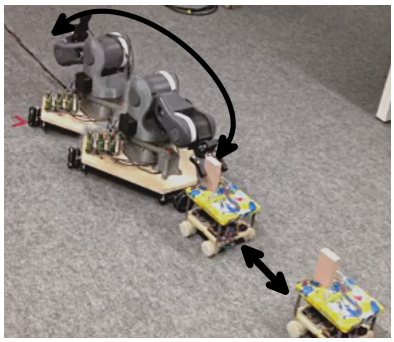


(a) The cart and arm robots attached together.

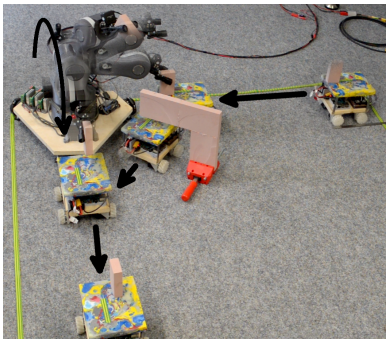


(b) The carrier robot.

Figure 4 Robots used in our experiments.



(a) Handover.



(b) Underpass.

Figure 5 Composite images of the experiments. (a) For handover, a cart containing an object moves close to the cart with the attached arm. The arm picks up the object. (b) For underpass, the carrier containing an object moves near the underpass. The arm picks up the object. The carrier moves under the underpass and moves close to the arm. The arm places the object on the carrier.

Composite images (combination of multiple frame of the video) are shown in Figure 5. The carts implement motion explicitly using the motion primitives (move straight, strafe, rotate). For instance, when going around the cart in the second experiment, the carrier executes rotate, move straight, rotate, strafe. In the model of the resources, we exclude the gripper from the footprint and we do not model the objects gripped (gripping is a collision). For the environment, we model obstacles as regions of $\mathbb{R}^3$ and also test for collision against these regions.

Table 3 shows the size of the programs in the language of PGCD (sum for all the robots) and the size of the global specifications. As part of the program we include a description of the environment which specifies the initial states of the robots and the obstacles used for additional collision checks. Finally, we show the number of verification conditions (#VCs) generated during the subtyping and the checks for joint trajectories. The total running time includes all the steps, i.e., checking the global specification, projection, typing, the existence of joint trajectories, and the absence of collision. The running time is dominated by the check on trajectories and collisions. The motion primitives (implementation and specification) are taken from PGCD without any change and represent around 1K lines of codes for all three robots.

Table 3 Programs, Specification, and Checks.

Experiment	Program (LoC)	Specification (LoC)	#VCs	Time (sec.)
Handover	22	12	141	38
Underpass	29	22	302	56

Compared to the verification results presented with PGCD [4, Section 5], we have roughly a 2× speed-up. The reason is that PGCD is used model-checking instead of global/local types. The motion session calculus makes it possible to have an abstract global specification which is easier to check.

In conclusion, our evaluation demonstrates that session types allow the specification of non-trivial co-ordination tasks between multiple robots with reasonable effort, while allowing automated and compositional verification.

7 Related Work

There is considerable interest in the robotics community on designing modular robotic components from higher-level specifications [32, 19]. However, most of this work has focused on descriptions for the physical and electronic design of components or on generating plans from higher level specifications rather than on language abstractions and types to reason about concurrency and motion. The interaction between concurrency and dynamics, and the use of automated verification techniques were considered in PGCD [4]. Our work takes PGCD as a starting point and formalises a compositional verification methodology through session types.

At the specification level, hybrid process algebras and other models of hybrid systems [1, 41, 7, 38] can model concurrent hybrid systems. However, these papers do not provide a direct path to implementation. Hybrid extensions to synchronous reactive languages [6, 5] describe programs which interact through events and control physical variables. Most existing verification methodologies for these programs rely on global model checking rather than on types. Our choice of session types is inspired by efficient type checking but also as the basis for describing interface specifications for components.

Extensions and applications of multiparty session types have been proposed in many different settings. See, e.g. [27, 3, 17]. We discuss only most related work. The work [9] extends multiparty session types with time, to enable the verification of realtime distributed systems. This extension with time allows specifications to express properties on the causalities of interactions, on the carried data types, and on the times in which interactions occur. The projected local types correspond to Communicating Timed Automata (CTA). To ensure the progress and liveness properties for projected local types, the framework requires several additional constraints on the shape of global protocols, such as feasibility condition (at any point of the protocol the current time constraint should be satisfiable for any possible past) and a limitation to the recursion where in the loop, the clock should be always reset. The approach is implemented in Python in [34] for runtime monitoring for the distributed system. Later, the work in [8] develops more relaxed conditions in CTAs, and applies them to synthesise timed global protocols. Unlike our work, no type checking for processes is studied in [8]. The main difference from [9, 34, 8] is that our approach does not rely on CTAs and is more specific to robotics applications where the verification is divided into the two layers; (1)

a simple type check for processes with motion primitives to ensure communication deadlock-freedom with global synchronisations; and (2) additional more refined checks for trajectories and resources. This two layered approach considerably simplifies our core calculus and typing system in Section 4, allowing to verify more complex scenarios for robotics interactions.

8 Conclusion

We have outlined a unifying programming model and typing discipline for communication-centric systems that sense and actuate the physical world. We work in the framework of multiparty session types [25, 26], which have proved their worth in many different scenarios relating to "pure" concurrent software systems. We show how to integrate motion primitives into a core calculus and into session types. We demonstrate how multiparty session types are used to specify correct synchronisation among multiple participants: we first provide a basic progress guarantee for communications and synchronisation by motion primitives, which is useful to extend richer verification related to trajectories.

At this point, our language is a starting point and not a panacea for robotics programming. Decoupling specifications into parallel and/or sequential tasks and using distributed controllers assumes "loosely coupled dynamics." In some examples, such as a multiple cart/arm co-ordination control, it may not be easy to assume a purely distributed control strategy based on independent motion primitives. We are thus exploring simultaneous concurrent programming and distributed controller and co-ordinator synthesis. As an example, assume that we have two cart/arm compositions which should lift one object together. In particular we can assume that lifting the object with only one arm would cause the cart/arm compositions to tilt over, which generates a strong coupling between all components during the coordinated lift of the object. Our framework allows to easily synchronise all the components. However, in any realistic scenario a robust controller 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, may be too weak or incur too much communication and bandwidth overhead for a real implementation.

Going in this direction, we need a better way to integrate specifications of controllers (motion primitives) and their robustness. This would also enable a more realistic non-synchronous model for the communication [31] and, after checking some robustness condition on the controller, rigorously show that the synchronous idealised model is equivalent to the more realistic model, i.e., considering delay in the communication as disturbances for the motion primitives. We also plan to tackle channel passing. The challenge is that the physical world (time and space) is hard to isolate: for instance, time is an implicit synchronisation which occurs at the same time across all sessions.

Finally, robotics applications manipulate physical state and time as *resources*. An interesting open question is how resource-based reasoning techniques such as separation logics for concurrency [36, 28] can be repurposed to reason about separation of components in space and time.

References

1 R. Alur and T.A. Henzinger. Modularity for Timed and Hybrid Systems. In *CONCUR '97: Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer, 1997.

2 R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, pages 163–178. Springer, 1998.

3 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Denielou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *FTPL*, 3(2-3):95–230, 2016.

4 Gregor B. Banusic, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien Zufferey. PGCD: robot programming and verification with geometry, concurrency, and dynamics. In Xue Liu, Paulo Tabuada, Miroslav Pajic, and Linda Bushnell, editors, *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 57–66. ACM, 2019. doi:10.1145/3302509.3311052.

5 Kerstin Bauer and Klaus Schneider. From synchronous programs to symbolic representations of hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 41–50. ACM, 2010. doi:10.1145/1755952.1755960.

6 Albert Benveniste, Timothy Bourke, Benoît Caillaud, Jean-Louis Colaço, Cédric Pasteur, and Marc Pouzet. Building a Hybrid Systems Modeler on Synchronous Languages Principles. *Proceedings of the IEEE*, 106(9):1568–1592, 2018. doi:10.1109/JPROC.2018.2858016.

7 J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2):215–280, 2005. Process Algebra. doi:10.1016/j.tcs.2004.04.019.

8 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *26th International Conference on Concurrency Theory*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl, 2015.

9 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In *25th International Conference on Concurrency Theory*, volume 8704 of *LNCS*, pages 419–434. Springer, 2014.

10 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *SFM*, volume 9104 of *LNCS*, pages 146–178. Springer, 2015.

11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *ESOP 2012 - European Symposium on Programming*. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

12 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

13 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.

14 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. In *PLACES*, volume 203 of *EPTCS*, pages 29–43, 2015. doi:10.4204/EPTCS.203.3.

15 Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. Denotational and Operational Preciseness of Subtyping: A Roadmap. In *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, volume 9660 of *LNCS*, pages 155–172. Springer, 2016.

16 Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013. doi:10.1007/978-3-642-38574-2_14.

17 Simon Gay and Antonio Ravera, editors. *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

18 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

19 Sehoon Ha, Stelian Coros, Alexander Alspach, James M. Bern, Joohyung Kim, and Katsu Yamane. Computational Design of Robotic Devices From High-Level Motion Specifications. *IEEE Trans. Robotics*, 34(5):1240–1251, 2018. doi:10.1109/TRO.2018.2830419.

20 Thomas A. Henzinger. Sooner is Safer Than Later. *Inf. Process. Lett.*, 43(3):135–141, 1992. doi:10.1016/0020-0190(92)90005-G.

21 Thomas A. Henzinger. The Theory of Hybrid Automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 278–292. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561342.

22 G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi:10.1109/32.588521.

23 Kohei Honda. Types for Dyadic Interaction. In *CONCUR'93*, pages 509–523, 1993.

24 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998. doi:10.1007/BFb0053567.

25 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM Press, 2008.

26 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of ACM*, 63:1–67, 2016.

27 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1), 2016. doi:10.1145/2873052.

28 R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL 15*, pages 637–650. ACM, 2015.

29 Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 395–409. Springer, 2013.

30 Dimitrios Kouzapas and Nobuko Yoshida. Globally Governed Session Semantics. *Logical Methods in Computer Science*, 10(4), 2015.

31 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 221–232. ACM, 2015.

32 A.M. Mehta, N. Bezzo, P. Gebhard, B. An, V. Kumar, I. Lee, and D. Rus. A Design Environment for the Rapid Specification and Fabrication of Printable Robots. *Experimental Robotics*, pages 435–449, 2015.

33 Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

34 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

35 Pierluigi Nuzzo. *Compositional Design of Cyber-Physical Systems Using Contracts*. PhD thesis, EECS Department, University of California, Berkeley, August 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html.

36 P.W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.

37 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

38 A. Platzer. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer, 2010.

39 K. V. S. Prasad. A Calculus of Broadcasting Systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995. doi:10.1016/0167-6423(95)00017-8.

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

41 W.C. Rounds and H. Song. The Phi-Calculus: A Language for Distributed Control of Reconfigurable Embedded Systems. In *HSCC*, pages 435–449. Springer, 2003.

42 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.

43 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994. doi:10.1007/3-540-58184-7_118.