Multiparty Motion Coordination: From Choreographies to Robotics Programs

RUPAK MAJUMDAR, Max Planck Institute for Software Systems, Germany
NOBUKO YOSHIDA, Imperial College London, United Kingdom
DAMIEN ZUFFEREY, Max Planck Institute for Software Systems, Germany

We present a programming model and typing discipline for complex multi-robot coordination programming. Our model encompasses both synchronisation through message passing and continuous-time dynamic motion primitives in physical space. We specify *continuous-time motion primitives* in an assume-guarantee logic that ensures compatibility of motion primitives as well as collision freedom. We specify global behaviour of programs in a *choreographic* type system that extends multiparty session types with jointly executed motion primitives, predicated refinements, as well as a *separating conjunction* that allows reasoning about subsets of interacting robots. We describe a notion of *well-formedness* for global types that ensures motion and communication can be correctly synchronised and provide algorithms for checking well-formedness, projecting a type, and local type checking. A well-typed program is *communication safe*, *motion compatible*, and *collision free*. Our type system provides a compositional approach to ensuring these properties.

We have implemented our model on top of the ROS framework. This allows us to program multi-robot coordination scenarios on top of commercial and custom robotics hardware platforms. We show through case studies that we can model and statically verify quite complex manoeuvres involving multiple manipulators and mobile robots—such examples are beyond the scope of previous approaches.

CCS Concepts: • **Computer systems organization → Robotics**; • **Software and its engineering → Co-operating communicating processes**; **Formal software verification**.

Additional Key Words and Phrases: Robotics, Message-passing, Session Types and Choreography, Continuous-time Motion Primitives.

ACM Reference Format:
Rupak Majumdar, Nobuko Yoshida, and Damien Zufferey. 2020. Multiparty Motion Coordination: From Choreographies to Robotics Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 134 (November 2020), 30 pages. https://doi.org/10.1145/3428202

1 INTRODUCTION

Modern robotics applications are often deployed in safety- or business-critical applications and formal specifications and reasoning about their correct behaviours is a difficult and challenging problem. These applications tightly integrate computation, communication, control of physical dynamics, and geometric reasoning. Developing programming models and frameworks for such

Authors' addresses: Rupak Majumdar, Max Planck Institute for Software Systems, Kaiserslautern, Germany, rupak@mpi-sws.org; Nobuko Yoshida, n.yoshida@imperial.ac.uk, Imperial College London, Computing, 180 Queen's Gate, London, SW7 2AZ, United Kingdom; Damien Zufferey, Max Planck Institute for Software Systems, Kaiserslautern, Germany, zufferey@mpi-sws.org.

2475-1421/2020/11-ART134
https://doi.org/10.1145/3428202

applications has been long noted as an important and challenging problem in robotics [Lozano-Pérez 1983], and yet very little support exists today for compositional specification of behaviours or their static enforcement.

In this paper, we present a programming model for concurrent robotic systems and a type-based approach to statically analyse programs. Our programming model uses *choreographies*—global protocols which allow implementing distributed and concurrent components without a central control—to compositionally specify and statically verify both message-based communications and jointly executed motion between robotics components in the physical world. Our choreographies extend *multiparty session types* [Honda et al. 2008] with *dynamic motion primitives*, which represent pre-verified motions that the robots can execute in the physical world. We compile well-typed programs into robotics platforms. We show through a number of nontrivial usecases how our programming model can be used to design and implement correct-by-construction, complex, robotic applications on top of commercial and custom-build robotic hardware.

Our starting point is the theory of motion session types [Majumdar et al. 2019], which forms a type discipline based on global types with simple, discrete-time motion primitives. For many applications, we found this existing model too restrictive: it requires that all components agree on a pre-determined, global, discrete clock and it forces *global synchronisation* among all robots at the fixed interval determined by the "tick" of the global clock. For example, two robots engaged in independent activities in different parts of a workspace must nevertheless stop their motion and synchronise every tick. This leads to communication-heavy programs in which the programmer must either pick a global clock that ensures every motion primitive can finish within one tick (making the system very slow) or that every motion primitive is interruptible (all robots stop every tick and coordinate). Our new model enhances the scope and applicability of motion sesstion types to robots: we go from the global and discrete clock to *continuous behaviours* over time, allowing complex synchronisations as well as *frame separation* between independent subgroups of robots. That is, the programmer does not need to think about global ticks when writing the program. Instead, they think in terms of motion primitives and the type system enforces that concurrent composition of motions is well-formed and that trajectories exist in a global timeline.

Reasoning simultaneously about dynamics and message-based synchronisation is difficult because time is global and can be used as an implicit broadcast synchronisation mechanism. The complexity of our model arises from the need to ensure that every component is simultaneously ready to let time progress (through dynamics) or ready to send or receive messages (property *communication safety*). At the same time, we must ensure that systems are able to correctly execute motion primitives (property *motion compatibility*); and jointly executed trajectories are separated in space and time (property *collision freedom*). Our verification technique is *static*: if a program type checks, then every execution of the system satisfies communication safety, motion compatibility, and collision freedom. We manage the complexity of reasoning about the interplay between dynamics and concurrency through a separation of concerns.

First, we specify *continuous-time trajectories* as *motion primitives*. Since the dynamics of different components can be coupled, the proof system uses an *assume-guarantee proof system* [Abadi and Lamport 1993; Chandy and Misra 1988; Jones 1983; Nuzzo et al. 2015] on continuous time processes to relate an abstract motion primitive to the original dynamics. The assume guarantee contracts decouple the dynamics. Additionally, the proof system also checks that trajectories ensure disjointness of the use of geometric space over time.

Second, we interpret choreographic specifications in continuous time, and extend the existing formalism in [Majumdar et al. 2019] with *predicate refinements*—to reason about permissions (i.e., what parts of the state space an individual robot can access without colliding into a different robot). This is required for motion compatibility and collision freedom. We also introduce a *separating* *conjunction* operator, written $*$, to reason about disjoint frames. The combination of message passing and dynamics makes static verification challenging. We introduce a notion of *well-formedness* of choreographies that ensures motion and communication can be synchronised and disallows, e.g., behaviours when a message is blocked because a motion cannot be interrupted. We give an algorithm for checking well-formedness using a dataflow analysis on choreographies. We show that well-formed types can be projected on to end-components and provide a local type checking that allows refinement between motion primitives.

Checking compatibility and collision freedom reduces to validity queries in the underlying logic. Interestingly, the separating conjunction allows subgroups of robots to interact—through motion and messages—disjointly from other subgroups; this reduces the need for global communication in our implementations.

We compile well-typed programs into programs in the PGCD [Banusic et al. 2019] and ROS [Quigley et al. 2009] frameworks. We have used our implementation to program and verify a number of complex robotic coordination and manipulation tasks. Our implementation uses both commercial platforms (e.g., the Panda 7DOF manipulator, the BCN3D MOVEO arm) and custom-built components (mobile carts). In our experience, our programming model and typing discipline are sufficient to specify quite complex manoeuvres between multiple robots and to obtain verified implementations. Moreover, the use of choreographies and the separating conjunction are crucial in reducing verification effort: without the separating conjunction, verification times out on more complex examples. Our implementation uses SMT solvers to discharge validity queries arising out of the proof system; our initial experience suggests that while the underlying theories (non-linear arithmetic) are complex, it is possible to semi-automatically prove quite involved specifications.

Contributions and Outline. This paper provides a static compositional modelling, verification, and implementation framework through behavioural specifications for concurrent robotics applications that involve reasoning about message passing, continuous control, and geometric separation. We manage this complexity by decoupling dynamics and message passing and enables us to specify and implement robotic applications on top of commercial and custom-build robotic hardware. Our framework coherently integrates programming languages techniques, session type theories, and static analysis techniques; this enables us to model *continuous behaviours* over time in the presence of complex synchronisations between independent subgroups of robots. We extend the global types in [Majumdar et al. 2019] to *choreographies* including key constructs such as framing and predicate refinements, together with separation conjunctions, and integrate the typing system with an assume-guarantee proof system. We implement our verification system using a new set of robots (extending [Banusic et al. 2019; Majumdar et al. 2019]) in order to program and verify complex coordinations and manipulation tasks.

The rest of the paper is organised as follows.

- Sec. 2**:** We first motivate our design for robotics specifications, and explain the correctness criteria – *communication safety*, *motion compatibility*, and *collision freedom* with an example.
- Sec. 3: We introduce a *multiparty motion session calculus* with *dynamic abstract motion primitives*; we provide an assume guarantee proof system to construct abstract motion primitives for maintaining geometric separation in space and time for concurrently executed motion primitives. We then prove Theorem 3.5 (*Motion Compatibility*).
- Sec. 4: We provide *choreographic specifications* enriched with dynamic motion primitives and separation operator, and define their dataflow analysis. We propose a typing system and prove its main properties, Theorem 4.12 (*Subject Reduction*), Theorem 4.13 (*Communication and Motion Progress* and *Collision-free Progress*).

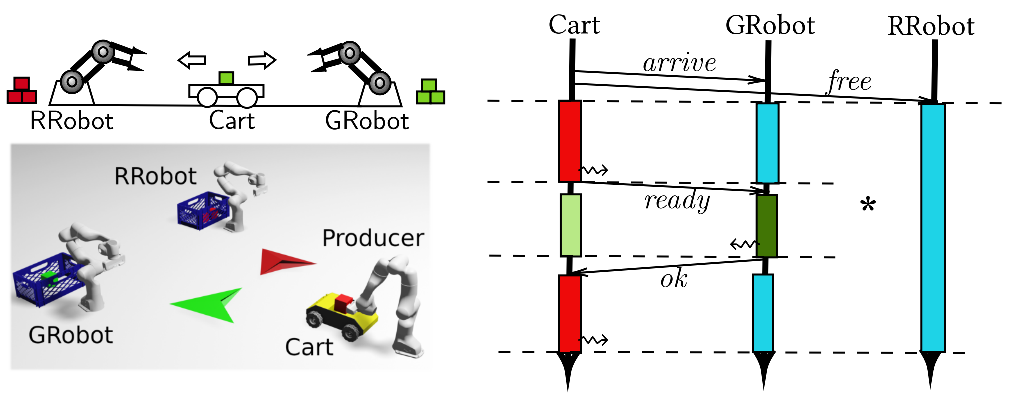


Fig. 1. An example with a cart and two robot arms (left: schematic on top, actual robots used in experiments in Sec. 5 bottom) and a partial message sequence for one behavior of the system (right). Message exchanges take zero time but are shown with downward sloping arrows for readability. The colored boxes denote motion primitives and their execution allows physical time to pass. The light blue box denotes the motion primitive "work" (in Figure 2), red denotes "m_move", light green "m_idle", dark green "pick". The dotted horizontal lines show synchronisation of global time across components. The use of the separating conjunction "$*$" ensures that time need not be synchronised with the red robot at intermediate synchronisations between the cart and the green robot. At each point, at most one motion primitive is non-interruptible (shown with "$\rightsquigarrow$")

$$\mu t.\left(\begin{array}{l}\left(\begin{array}{l}\boxed{\mathsf{Cart} \to \mathsf{GRobot} : \mathit{arrive}.\mathsf{Cart} \to \mathsf{RRobot} : \mathit{free}}\,. \\ \left(\left(\boxed{\begin{array}{l}\mathsf{dt}\langle \mathsf{Cart} : \mathsf{m\_move}^{\rightsquigarrow}(\mathsf{point}), \mathsf{GRobot} : \mathsf{work}^{\lightning}\rangle. \\ \mathsf{Cart} \to \mathsf{GRobot} : \mathit{ready}. \\ \mathsf{dt}\langle \mathsf{Cart} : \mathsf{m\_idle}^{\lightning}, \mathsf{GRobot} : \mathsf{pick}^{\rightsquigarrow}\rangle. \\ \mathsf{GRobot} \to \mathsf{Cart} : \mathit{ok}. \\ (\mathsf{dt}\langle \mathsf{Cart} : \mathsf{m\_move}^{\rightsquigarrow}(\mathsf{point})\rangle * \mathsf{dt}\langle \mathsf{GRobot} : \mathsf{work}^{\lightning}\rangle)\end{array}}\right) * \mathsf{dt}\langle \mathsf{RRobot} : \mathsf{work}^{\lightning}\rangle\right)\end{array}\right) \\ + \\ (\quad \text{as the above swapping GRobot with RRobot} \quad)\end{array}\right).t$$

Fig. 2. Global choreography for the example

<ins>Sec. 5</ins>: We describe our implementation, evaluation, and case studies. Our evaluation demonstrates that our framework allows specifying complex interactions and verifying examples beyond the scope of previous work.

<ins>Sec. 6 and Sec. 7</ins>: We discuss related work and conclude with a number of open challenges in reasoning about robotics applications.

Putting it all together, we obtain a compositional specification and implementation framework for concurrent robotics applications. Our paper is a starting point rather than the final word on robot programming. Indeed, we make many simplifying assumptions about world modeling, sensing and filtering, and (distributed) feedback control and planning. Even with these simplifications, the theoretical development is non-trivial. With a firm understanding of the basic theory, we hope that future work will lift many of the current limitations.

The full version [Majumdar et al. 2020] includes the detailed proofs.

2 MOTIVATING EXAMPLE

In this section, we show a coordination example and use it to motivate the different choices of our programming model and choreography specifications. We start with a simple protocol. Consider a scenario where a cart shuttles (based on some unmodeled criterion) between two robotic arms, "red" and "green" (corresponding, e.g., to two different processing units). In each round, the cart lets the robots know of its choice and then moves to the chosen robot. On reaching its destination, the cart signals to the arm that it has arrived, and waits for the arm to finish processing. Meanwhile, the other arm can continue its own work independently. When the arm finishes its processing, it signals the cart that processing is complete. The cart moves back and then repeats the cycle. Figure 1 shows a schematic of the system as well as a sample sequence of messages in each cycle.

Multiparty Motion Choreographies. We specify the global behaviour of a program using *choreographies*, which describe the allowed sequence of global message exchanges and joint motion primitives as types for programs. The choreography for our example is shown in Figure 2. It extends session types [Bocchi et al. 2015, 2019, 2014] with joint motion primitives similarly to motion session types [Majumdar et al. 2019] and a *separating conjunction* $*$.

(1) Messages and motion primitives. The choreography in Figure 2 is a ***recursion*** ($\mu \mathrm{t}.G$), that makes a ***choice*** ($G_1 + G_2$) between two possibilities, corresponding to the cart interacting with the green or with the red robots (blue box). ***Message exchange*** is specified by $\mathrm{p} \to \mathrm{q} : \ell$ which is a flow of a message labeled ℓ from process p to process q. ***Joint motion primitives*** $\mathrm{dt}\langle \mathrm{p} : a, \mathrm{p}' : a', \cdots \rangle$ specify that the processes p, p', etc. jointly execute motion primitives a, a', etc. for the same amount of time. Global time is a global synchroniser; the type system makes sure that there is a consistent execution that advances time in the same way for every process.

With just the construct for joint motion primitives, every component is forced to globally synchronise in time, preventing the robots to work independently in physically isolated spaces. We would like to specify, for example, that when the cart and the green robot GRobot is interacting, the red robot RRobot can independently perform its work without additional synchronisations.

(2) Separating conjunctions. To overcome this shortcoming, we introduce the ***separating conjunction*** $*$ (a novel addition from previous session types [Bocchi et al. 2015, 2019, 2014; Majumdar et al. 2019], explained below), which decomposes the participants into *disjoint* subgroups, in terms of communication, dynamics, and use of geometric space over time, each of which can proceed independently until a merge point.

The core interaction between the cart and an arm happens within the green box. Let us focus on the interaction when the green robot is picked. In this case, the cart moves to the green robot (while the green arm performs internal work), then synchronises with a *ready* message, and then idles while the green arm picks an object from the cart. When the arm is done, it sends an *ok* message, the cart moves back while the arm does internal work, and the protocol round ends. Meanwhile, the red robot can independently perform its work without synchronising with either the cart or the green robot. The other branch of + is identical, swapping the roles of the red and the green robots.

The separating conjunction makes it easy to specify physically and logically *independent* portions of a protocol. Without it, we would be forced to send a message to the other (red) arm every time the cart and the green arm synchronised, even if the red arm proceeded independently. This would be required because time is global, and we would have to ensure that all motion primitives execute for exactly the same duration. Obviously, this leads to unwanted global synchronisations.

(3) Interruptible and non-interruptible motions. Our type system ensures that messages and motions alternate in the correct way. This requires differentiating between motion primitives that are ***interruptible*** (marked by $\lightning$) from those that are not interruptible. An interruptible motion primitive is one that can be interrupted by a message receipt. For example, m_idle or work in our example are interruptible. A ***non-interruptible*** motion primitive (marked by $\rightsquigarrow$), on the other hand, should not be "stopped" by an external message. In order to ensure proper synchronisation between motion primitives and messages, we annotate motion primitives with $\lightning$ or $\rightsquigarrow$, which specify whether a motion can be interrupted by a message receipt or not. The type system checks that there is, at any time, at most one non-interruptible motion primitive, and moreover, the process executing that motion is the next one to send a synchronising message. This ensures that motion is compatible with synchronisation.

The assumption that there is at most one non-interruptible motion primitive is a restriction; it rules out complex multi-robot maneuvres. However, there are already many "loosely coupled" systems that satisfy the assumption and it allows us to develop the (already complicated) theory without introducing distributed feedback control strategies to the mix. We leave lifting the assumption to future work.

Processes: Motion Primitives + Messages. We model robotic systems as concurrent processes in a variant of the session π-calculus for multiparty interactions [Yoshida and Gheri 2020]. Our calculus (defined in Sec. 3) abstracts away from the sequential operations and only considers the *synchronisation behaviour* and the *physical state changes*. Synchronisation is implemented through synchronous message exchanges between individual physical components (participants). Between message exchanges, each process executes *motion primitives*: controller code that implements an actual robot motion. Motion primitives affect state changes in the physical world.

Correctness Criteria. Our type system prevents a well-typed implementation from "going wrong" in the following ways.

Communication Safety. We provide a choreography type system to ensure that programs are *communication safe* and *deadlock free*: a component does not get stuck sending a message with no recipient or waiting for a non-existent message. Ensuring communication safety is trickier for robotics programs because components may not be ready to receive messages because they are in the middle of executing a motion primitive that cannot be interrupted.

Motion Compatibility. We check that programs are able to *jointly* execute motion primitives. The controllers implementing motion primitives implement control actions for coupled dynamical systems and we have to check that controllers for different components can be run together.

Collision Freedom. Components occupy 3D space; a correct implementation must ensure no geometric collision among objects. For example, we must ensure that the robot arms do not hit the cart when the cart is moving in their workspace. Our motion compatibility check ensures the geometric footprint of each component remains disjoint from others at all points in time.

Type Checking. The type checking algorithm has three parts. (1) First, we use an ***assume guarantee proof system*** to ensure jointly executed motion primitives are ***compatible***: they allow a global trajectory for all the robots in the system and ensure there is no collision between components. (2) Second, we introduce a notion of ***well-formedness*** for choreographies. Well-formed choreographies ensure motion and message exchanges can be globally synchronised and that joint trajectories of the system are well-defined and collision free. (3) Finally, as in standard session types, we ***project a global choreography to its end-points***, yielding a local type that can be checked for each process. The challenge is to manage the delicate interplay between time, motion, message exchanges, choices and the separating conjunction. This makes the well-formedness check more sophisticated than other choreography type systems.

3 MULTIPARTY MOTION SESSION CALCULUS

We now describe the motion session calculus, extending from [Majumdar et al. 2019].

3.1 Syntax

Physical Components and Processes. We assume a fixed static set $\mathbb{P}$ of *physical components* (ranged over by p, q,...), which are often called *participants* or *roles*; these are the different components that constitute the overall system. Each physical component executes a software process, which takes care of communication, synchronisation and motion. We describe the motion session calculus, which forms the core of the software process.

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. A *motion primitive* (a, b, ...) is an abstraction of underlying physical trajectories; for the moment, consider a motion primitive simply as a name and describe its semantics later. We use the notation $\mathsf{dt}\langle a\rangle$ to represent that a motion primitive executes and time elapses. 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.

The *processes* ($P, Q, R, \ldots$) of the multiparty motion session calculus are defined by:

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

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. Motion primitives are indicated by $\mathsf{dt}\langle a\rangle$; the dt denoting that real time progresses when a motion primitive a is executed. The conditional process if e then P 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 $\mu\mathsf{x}.P$ is a recursive process. Note that our processes do not have a null process: this is because a physical component does not "disappear" when the program stops. Instead, we model inaction with an iteration of some default motion primitive.

Example 3.1 (Processes). The processes for the cart and the two robots from Figure 1 are given as:

$$\begin{array}{ll}
\mathsf{Cart}: & \mathsf{RRobot},\\
\mu\mathsf{x}.(\mathsf{GRobot}!\mathit{arrive}.\mathsf{RRobot}!\mathit{free}. & \mathsf{GRobot}:\\
\quad \mathsf{dt}\langle\mathsf{m\_move}(\text{co-ord of }\mathsf{GRobot})\rangle.\mathsf{GRobot}!\mathit{ready}. & \mu\mathsf{x}.(\mathsf{Cart}?\mathit{arrive}.\mathsf{dt}\langle\mathsf{work}\rangle.\mathsf{Cart}?\mathit{ready}.\\
\quad \mathsf{dt}\langle\mathsf{m\_idle}\rangle.\mathsf{GRobot}?\mathit{ok}.\mathsf{dt}\langle\mathsf{m\_move}(\mathsf{base})\rangle.\mathsf{x} & \quad \mathsf{dt}\langle\mathsf{pick}\rangle.\mathsf{Cart}!\mathit{ok}.\mathsf{dt}\langle\mathsf{work}\rangle).\mathsf{x}\\
\quad + \text{ symmetrically for } \mathsf{RRobot}\ ) & \quad + (\mathsf{Cart}?\mathit{free}.\mathsf{dt}\langle\mathsf{work}\rangle).\mathsf{x}
\end{array}$$

Note that for both processes, the recursion ensures that the program does not terminate. Motion primitives for the cart involve moving to various locations m_move(pos) and idling at a location (m_idle), and those for the robots involve doing some (unspecified) work work or picking items off the cart pick (internally, these motion primitives would involve motion planning and inverse kinematics for the robot arms). Next, we describe the modelling and representation of motion primitives in more detail.

Physical Variables and Footprint. Motion primitives make the robot "move" in the physical world. Each motion primitive represents an abstraction of an underlying controlled dynamical system, such as the controller for the robot arm or the controller for a cart. The dynamical system changes underlying physical state (such as the position and orientation of the arm or the position and velocity of the cart). The dynamics can be coupled: for example, if an arm is mounted on a cart, then the motion of the cart is influenced by the mass and position of the arm.

We model the underlying physical system using *physical variables*, and we partition these into state variables X (dynamical variables read and controlled by the component), input variables W (dynamical variables read by the component, whose values are provided by the environment). The specification of a motion primitive will constrain the values of these variables over time. We make the physical variables clear by writing $\mathsf{p} \triangleleft \langle X, W; P\rangle$ for a physical component $\mathsf{p} \in \mathbb{P}$ with state and input variables X and W, respectively, which executes the process P.

Each physical component $\mathsf{p} \in \mathbb{P}$ has a *geometric footprint* $\mathsf{geom}(\mathsf{p})$ associated with it. This represents the physical space occupied by the component, and will be used to ensure that two components do not collide. The footprint is a function from valuations to variables in X to a subset of $\mathbb{R}^3$. It describes an overapproximation of the space occupied by the component as a function of the current state. Note that the footprint can depend on the state.

Example 3.2. Let x and v denote the position and the velocity of the cart, respectively, restricting the discussion only on one axis. Thus, $X = \{x, v\}$. If we assume there are no external influences on the cart, we can take $W = \emptyset$. The footprint provides a bounding box around the cart as it moves. If the cart has dimensions (l, w, h), the footprint at the location (x, y, z) is

$$\{(\mathsf{x},\mathsf{y},\mathsf{z}) \mid x - \frac{l}{2} \leq \mathsf{x} \leq x + \frac{l}{2} \wedge 0 \leq \mathsf{y} \leq h \wedge z - \frac{w}{2} \leq \mathsf{z} \leq z + \frac{w}{2}\} \tag{1}$$

Composition. We now define parallel composition of physical components. Composition of physical components ensure the following aspects. First, like process calculi, parallel composition provides a locus for message exchange: a physical component can send messages to, or receive messages from, another one. Second, composition connects physical state variables of one component to the physical input variables of another—this enables the coupling of the underlying dynamics.

To ease reasoning about connections between physical variables, we assume that the components in a composition have disjoint sets of physical and logical variables, and connections occur through syntactic naming of input variables. Thus, a component with an input variable x gets its value from the unique component that has a physical state variable called x. Hence, there is no ambiguity in forming connections. We refer to the variables of p as $\mathsf{p}.X$ and $\mathsf{p}.W$.

We define a *multiparty session* as a parallel composition of pairs of participants and processes:

$$M \quad ::= \quad \mathsf{p} \triangleleft \langle X, W; P\rangle \mid M \mid M$$

with the intuition that process P plays the role of participant p, and can interact with other processes playing other roles in M. A multiparty session is *well-formed* if all its participants are different, and for each input variable of each participant there is a syntactically unique state variable in a different participant so that connections between physical variables is well-defined. We consider only well-formed processes.

Example 3.3. For the example from Sec. 2, the multiparty session is

$$\mathsf{Cart}\triangleleft\langle\{x, v\}, \emptyset; \text{proc. for } \mathsf{Cart}\rangle \mid \mathsf{RRobot}\triangleleft\langle\cdot,\cdot; \text{proc. for } \mathsf{RRobot}\rangle \mid \mathsf{GRobot}\triangleleft\langle\cdot,\cdot; \text{proc. for } \mathsf{GRobot}\rangle$$

(we have omitted the physical variables for the robot arms for simplicity).

3.2 Motion Primitives

Before providing the operational semantics, we first consider how motion primitives are specified. Recall that motion primitives of a component $\mathsf{p} \triangleleft \langle X, W; P\rangle$ abstract a trajectory arising out of the underlying dynamics. A first idea is to represent a motion primitive as a pair $\langle \mathrm{Pre}, \mathrm{Post}\rangle$—this provides a precondition Pre that specifies the condition on the state under which the motion primitive can be applied and a postcondition Post that specifies the effect of applying the motion primitive on the physical state. However, this is not sufficient: the motion of two components can be coupled in time and the trajectory of a component depends on the inputs it gets from other components and, in turn, its trajectory influences the trajectories of the other components. Therefore, a motion primitive also needs to specify *assumptions* on its external inputs and *guarantees* it provides to other processes over time. These predicates are used to decouple the dynamics.

Our motion primitive has three more ingredients. The first is the *footprint*: the geometric space used by a process over time while it executes its trajectory and is used to ensure geometric separation between components. The second is a pair $(D, \ddagger)$ of a *time interval and an annotation*: the time interval bounds the *minimal* and *maximal times* between which a motion primitive is ready to communicate via message passing, and the $\ddagger$ annotation distinguishes between motion primitives that are interruptible by external messages from those that cannot be interrupted.

We assume preconditions and postconditions only depend on the state, while assumptions, guarantees, and footprints may depend on both the state and the elapsed time.

Definition 3.4 (Motion primitive). The specification for a motion primitive a of a physical process $\mathsf{p} \triangleleft \langle X, W; P\rangle$, written $a \vdash (\mathrm{Pre}, A, G, \mathrm{Post}, FP, D, \ddagger)$, consists of two predicates *precondition* Pre and *postcondition* Post over $X \cup W$, an *input assumption* A which is a predicate over $W \cup \{\mathsf{clock}\}$, an *output guarantee* G which is a predicate over $X \cup \{\mathsf{clock}\}$, a *footprint* FP which is a predicate over $X \cup \{\mathsf{x}, \mathsf{y}, \mathsf{z}, \mathsf{clock}\}$, a *duration* $D \in (\mathbb{R} \cup \{\infty\})^2$ which is a time interval, and $\ddagger \in \{\lightning, \rightsquigarrow\}$ which indicates if the motion primitive can be interrupted by an external message ($\lightning$) or not ($\rightsquigarrow$).

Given motion primitives a and a', we say a *refines* a', denoted by $a \preceq a'$, with $a \vdash (\varphi, A, G, \psi, FP, D, \ddagger)$ and $a' \vdash (\varphi', A', G', \psi', FP', D', \ddagger')$ iff (1) $\varphi \Rightarrow \varphi'$, (2) $A \Rightarrow A'$, (3) $G' \Rightarrow G$, (4) $\psi' \Rightarrow \psi$, (5) $FP' \subseteq FP$, (6) $\ddagger = \ddagger'$, and (7) either $\ddagger = \lightning$ and $D \subseteq D'$ or $\ddagger = \rightsquigarrow$ and $D' \subseteq D$.

3.3 Operational Semantics

The operational semantics is given as reduction rules relative to *stores* $\mathtt{X}, \mathtt{W}$ (Figure 3) that map physical variables to values. The semantics uses the standard structural rules defined in Figure 4.

We adopt some standard conventions regarding the syntax of processes and sessions. Namely, we will use $\prod_{i \in I} \mathsf{p}_i \triangleleft \langle X_i, W_i; P_i\rangle$ for $\mathsf{p}_1 \triangleleft \langle X_1, W_1; P_1\rangle \mid \ldots \mid \mathsf{p}_n \triangleleft \langle X_n, W_n; P_n\rangle$, where $I = \{1, \ldots, n\}$, or simply as $\prod_{i \in I} \mathsf{p}_i \triangleleft P_i$ when the physical variables are not important. We use infix notation for external choice process, e.g., instead of $\sum_{i \in \{1,2\}} \mathsf{p}?\ell_i(x).P_i$, we write $\mathsf{p}?\ell_1(x).P_1 + \mathsf{p}?\ell_2(x).P_2$. *The value v of expression* e *with physical state* $\mathtt{X}$ (notation $\mathsf{e} \downarrow_{\mathtt{X}} v$) is computed as expected. We assume that $\mathsf{e} \downarrow_{\mathtt{X}} v$ is effectively computable and takes logical "zero time."

For reduction rules that do not change the physical state, we omit writing the physical state in the rule. Time is global, and processes synchronize in time to make concurrent motion steps of the same (but not pre-determined) duration. Communication ([COMM]) is synchronous and puts together sends and receives. Rule [DEFAULT] selects the default branch. Rules for conditionals, communication without default motion and parallel composition are defined in a standard way [Dezani-Ciancaglini et al. 2016; Ghilezan et al. 2019a; Majumdar et al. 2019].

To define the operational semantics for motions, we extend the process syntax P with time-annotated motion primitives, $\mathsf{dt}\langle a@t\rangle$ for $t \geq 0$. Let us fix a component p and its motion primitive $a \vdash (\mathrm{Pre}, A, G, \mathrm{Post}, FP, D, \ddagger)$. Rules [TRAJ-BASE] and [TRAJ-STEP] set up trajectories for each motion primitive. We distinguish between interruptible ([INTERRUPT]) and non-interruptible ([NON-INTERRUPT]) motion primitives. Non-interruptible motion primitives are consumed by the process. Interruptible motion primitives consume the motion primitive on a message receipt. The parallel composition rule [M-PAR] requires a consistent global trajectory and ensures that when (physical)

$$[\textsc{recv}]\ \dfrac{j \in I}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W};\ \sum_{i\in I} \mathtt{q}?\ell_i(x).P_i + \mathsf{dt}\langle a\rangle.P\rangle \xrightarrow{\mathtt{q}?\ell_j(v)} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P_j\{v/x\}\rangle}$$

$$[\textsc{send}]\ \dfrac{\mathsf{e} \downarrow_{\mathtt{X},\mathtt{W}} v}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathtt{q}!\ell_j\langle \mathsf{e}\rangle.Q\rangle \xrightarrow{\mathtt{q}!\ell\langle v\rangle} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; Q\rangle} \qquad [\textsc{comm}]\ \dfrac{\mathtt{p} \triangleleft P \xrightarrow{\mathtt{q}!\ell\langle v\rangle} \mathtt{p} \triangleleft P' \qquad \mathtt{q} \triangleleft Q \xrightarrow{\mathtt{p}?\ell(v)} \mathtt{q} \triangleleft Q'}{\mathtt{p} \triangleleft P \mid \mathtt{q} \triangleleft Q \longrightarrow \mathtt{p} \triangleleft P' \mid \mathtt{q} \triangleleft Q'}$$

$$[\textsc{default}]\ \dfrac{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a\rangle.P\rangle \xrightarrow{\tau} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@0\rangle.P\rangle \qquad \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@0\rangle.P\rangle \xrightarrow{\mathsf{dt}\langle a\rangle, \xi, \nu, t} \mathtt{p} \triangleleft \langle \mathtt{X}', \mathtt{W}'; \mathsf{dt}\langle a@t\rangle.P\rangle}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W};\ \sum_{i\in I} \mathtt{q}?\ell_i(x).P_i + \mathsf{dt}\langle a\rangle.P\rangle \xrightarrow{\mathsf{dt}\langle a\rangle, \xi, \nu, t} \mathtt{p} \triangleleft \langle \mathtt{X}', \mathtt{W}'; \mathsf{dt}\langle a@t\rangle.P\rangle}$$

$$[\textsc{traj-base}]\ \dfrac{\mathrm{Pre}\{\mathtt{X}/X, \mathtt{W}/W, 0/\mathsf{clock}\} \quad A\{\mathtt{W}/W, 0/\mathsf{clock}\} \wedge G\{\mathtt{X}/X, 0/\mathsf{clock}\} \qquad \mathsf{geom}\,(\mathtt{p})\,(\mathtt{X}) \subseteq FP\{\mathtt{X}/X, 0/\mathsf{clock}\}}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a\rangle.P\rangle \xrightarrow{\tau} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@0\rangle.P\rangle}$$

$$[\textsc{traj-step}]\ \dfrac{\begin{array}{c}\xi : [t_1, t_2] \to \mathbb{R}^X, \nu : [t_1, t_2] \to \mathbb{R}^W \qquad \xi(t_1) = \mathtt{X}, \xi(t_2) = \mathtt{X}' \qquad \nu(t_1) = \mathtt{W}, \nu(t_2) = \mathtt{W}' \\ \forall t \in [t_1, t_2].\ A\{\nu(t)/W, t/\mathsf{clock}\} \wedge G\{\xi(t)/X, t/\mathsf{clock}\}\ \wedge \mathsf{geom}\,(\mathtt{p})\,(\xi(t)) \subseteq FP\{\mathtt{X}/X, t/\mathsf{clock}\}\end{array}}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@t_1\rangle.P\rangle \xrightarrow{\mathsf{dt}\langle a\rangle, \xi, \nu, t_2 - t_1} \mathtt{p} \triangleleft \langle \mathtt{X}', \mathtt{W}'; \mathsf{dt}\langle a@t_2\rangle.P\rangle}$$

$$[\textsc{t-conditional}]\ \dfrac{\mathsf{e} \downarrow_{\mathtt{X},\mathtt{W}} \mathsf{true}}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q\rangle \longrightarrow \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P\rangle} \qquad [\textsc{non-interrupt}]\ \dfrac{t \in D \qquad \ddagger = \rightsquigarrow \qquad \mathrm{Post}\{\mathtt{X}/X, \mathtt{W}/W, t/\mathsf{clock}\}}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@t\rangle.P\rangle \xrightarrow{\tau} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P\rangle}$$

$$[\textsc{interrupt}]\ \dfrac{t \in D \qquad \ddagger = \lightning \qquad \mathrm{Post}\{\mathtt{X}/X, \mathtt{W}/W, t/\mathsf{clock}\} \qquad \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P\rangle \xrightarrow{\mathtt{q}?\ell(v)} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P'\rangle}{\mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; \mathsf{dt}\langle a@t\rangle.P\rangle \xrightarrow{\mathtt{q}?\ell(v)} \mathtt{p} \triangleleft \langle \mathtt{X}, \mathtt{W}; P'\rangle}$$

$$[\textsc{m-par}]\ \dfrac{\exists \xi, \nu.\ \forall i, j \in I. \qquad i \neq j \Rightarrow \mathsf{disjoint}(a_i, a_j) \qquad \mathtt{p}_i \triangleleft \langle \mathtt{X}_i, \mathtt{W}_i; P_i\rangle \xrightarrow{\mathsf{dt}\langle a_i\rangle, \xi|_{X_i}, (\xi\cup\nu)|_{W_i}, t} \mathtt{p}_i \triangleleft \langle \mathtt{X}_i', \mathtt{W}_i'; P_i'\rangle}{\prod_{i\in I} \mathtt{p}_i \triangleleft \langle \mathtt{X}_i, \mathtt{W}_i; P_i\rangle \xrightarrow{\mathsf{dt}\langle a_i\rangle, t} \prod_{i\in I} \mathtt{p}_i \triangleleft \langle \mathtt{X}_i', \mathtt{W}_i'; P_i'\rangle}$$

$$[\textsc{r-par}\tau]\ \dfrac{M_1 \xrightarrow{\tau} M_2}{M_1 \mid M \xrightarrow{\tau} M_2 \mid M} \qquad [\textsc{r-par}]\ \dfrac{M_1 \longrightarrow M_2}{M_1 \mid M \longrightarrow M_2 \mid M} \qquad [\textsc{r-struct}]\ \dfrac{M_1' \equiv M_1 \xrightarrow{\alpha} M_2 \equiv M_2'}{M_1' \xrightarrow{\alpha} M_2'}$$

We omit [F-CONDITIONAL]. We use $\xrightarrow{\alpha}$ for any labelled transition relation or reduction ($\longrightarrow$).

Fig. 3. Operational semantics

$$[\textsc{s-rec}]\ \mu\mathtt{x}.P \equiv P\{\mu\mathtt{x}.P/\mathtt{x}\} \qquad [\textsc{s-multi}]\ P \equiv Q \Rightarrow \mathtt{p} \triangleleft P \mid M \equiv \mathtt{p} \triangleleft Q \mid M \qquad [\textsc{s-par 1}]\ M \mid M' \equiv M' \mid M \qquad [\textsc{s-par 2}]\ (M \mid M') \mid M'' \equiv M \mid (M' \mid M'')$$

Fig. 4. Structural congruence rules

time elapses for one process, it elapses equally for all processes. Here, $\mathsf{disjoint}(a_i, a_j)$ states that the footprints along the trajectory are disjoint: $a_i.FP\{\xi_i(t')/X, t'/\mathsf{clock}\} \cap a_j.FP\{\xi_j(t')/X, t'/\mathsf{clock}\} = \emptyset$ for all $t' \in [0, t]$.

We use $\longrightarrow^*$ for the reflexive transitive closure of $\longrightarrow$. We say a program state $\prod_i \mathtt{p}_i \triangleleft \langle \mathtt{X}_i, \mathtt{W}_i; P_i\rangle$ is *collision free* if $\mathsf{geom}\,(\mathtt{p}_i)\,(\mathtt{X}_i) \cap \mathsf{geom}\,(\mathtt{p}_j)\,(\mathtt{X}_j) = \emptyset$ for every $i \neq j$.

3.4 Joint Compatibility of Motion Primitives

Two motion primitives are *compatible* if they can be jointly executed. To decide compatibility, we compose the specifications using the following *assume-guarantee proof rule*:

$$[\textsc{AGcomp}]\quad \frac{\begin{array}{c}\exists FP_1, FP_2.\, G_1 \wedge G_2 \Rightarrow FP_1 \cap FP_2 = \emptyset \quad G_1 \wedge G_2 \Rightarrow FP_1 \cup FP_2 \subseteq FP \\ \ddagger_1 = \lightning \vee \ddagger_2 = \lightning \quad \ddagger_1 = \rightsquigarrow \Rightarrow D_1 \subseteq D_2 \quad \ddagger_2 = \rightsquigarrow \Rightarrow D_2 \subseteq D_1 \\ a_1 \leq \langle \varphi_1, A \wedge G_2, G_1, FP_1, \psi_1, D_1, \ddagger_1 \rangle \quad a_2 \leq \langle \varphi_2, A \wedge G_1, G_2, FP_2, \psi_2, D_2, \ddagger_2 \rangle\end{array}}{\langle \mathtt{p}_1 : a_1, \mathtt{p}_2 : a_2 \rangle \vdash (\varphi_1 \wedge \varphi_2, A, G_1 \wedge G_2, FP, \psi_1 \wedge \psi_2, D_1 \cap D_2, \ddagger_1 \oplus \ddagger_2)}$$

The $\oplus$ operator combines the interruptibility: $\lightning \oplus \lightning = \lightning$ and $\lightning \oplus \rightsquigarrow = \rightsquigarrow \oplus \lightning = \rightsquigarrow \oplus \rightsquigarrow = \rightsquigarrow$.

The [AGcomp] rule performs three checks. First, the check on footprints ensures that the motion primitives are disjoint in space (there is a way to find subsets FP_1 and FP_2 of the footprint FP so that the composed motion primitives are in these disjoint portions). Second, the guarantee of one motion primitive is used as the assumption of the other to check that they are compatible. Third, the checks on timing ensures that at most one process executes a non-interruptible motion primitive and the interruptible ones are ready before the non-interruptible one.

We say $\mathsf{dt}\langle(\mathtt{p}_i{:}a_i)\rangle$ is *compatible* from Pre if there exist G, Post, FP, and D such that $\prod_i \mathtt{p}_i : a_i \vdash (\mathrm{Pre}, \mathsf{true}, G, \mathrm{Post}, FP, D, \ddagger)$ is derivable using [AGcomp] repeatedly. Thus, compatibility checks that motion primitive specifications can be put together if processes start from their preconditions: first, the assumptions and guarantees are compatible; second, there is no "leftover" assumption; and third, the footprints of the motion primitives do not intersect in space. Compatibility provides the "converse" condition that allows joint trajectories in [M-par] to exist.

The next theorem formalizes what motion compatibility guarantees. Intuitively, motion compatibility means it is sufficient to check the compatibility of contracts to guarantee the existence of a joint trajectory, i.e., the execution is defined for all the components and the joint trajectory satisfies the composition of the contracts.

Theorem 3.5 (Motion Compatibility). *Suppose* $\mathsf{dt}\langle(\mathtt{p}_1 : a_1, \mathtt{p}_2 : a_2)\rangle$ *is compatible. For every* $t \in D$*, if there exist trajectories* $\xi_1, \xi_2, \nu_1, \nu_2$ *such that* $\mathtt{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle a_i \rangle, \xi_i, \nu_i, t} \mathtt{p}_i \triangleleft P_i'$ *for* $i \in \{1, 2\}$*, then there exist trajectories* $\xi : [0, t] \to \mathbb{R}^{X_1 \cup X_2}$, $\nu : [0, t] \to \mathbb{R}^{W_1 \cup W_2 \setminus (X_1 \cup X_2)}$ *such that* $\mathtt{p}_i \triangleleft P_i \xrightarrow{\mathsf{dt}\langle a_i \rangle, \xi|_{X_i}, \nu|_{W_i}, t} \mathtt{p}_i \triangleleft P_i'$ *and for all* $0 \leq t' \leq t$*, the footprints of* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *are disjoint:* $\mathsf{geom}\,(\mathtt{p}_1)\,(\xi(t')|_{X_1}) \cap \mathsf{geom}\,(\mathtt{p}_2)\,(\xi(t')|_{X_2}) = \emptyset$.

Proof. (Sketch.) The proof follows the AG rule by Henzinger et al. [2001]. Their proof relies on motion primitives having the following properties: prefix closure, deadlock freedom, and input permissiveness. Deadlock freedom, in this setting [Henzinger et al. 2001], means that if a precondition is satisfied then the trajectory must exist, and every execution that does not yet satisfy the postcondition can be prolonged. Input permissiveness states that a component cannot deadlock no matter how the environment decides to change the inputs. This condition is needed to do induction over time. Input permissiveness does not directly follow from the definition of our contracts. However, it holds when the environment changes the inputs in a way that is allowed by the assumptions. As [AGcomp] checks this condition and rejects the composition otherwise, we can reuse the same proof strategy. We also need to check the disjointness of footprints which is new in our model (needed by [M-par]). This is also checked by [AGcomp]. □

3.5 Examples for Motion Primitives, Compatibilities, and Environments

Motion primitives from dynamics. For the Cart in Sec. 2, we can derive motion primitives from a simple dynamical model $\dot{x} = v, \dot{v} = u$. Here, $X = \{x, v\}$ and $W = \emptyset$. For simplicity, assume the cart moves along a straight line (its x-axis) and that the control u can apply a fixed acceleration

$a_{\max}$ or a fixed deceleration $-a_{\max}$. Consider the motion primitive m_move which takes the cart from an initial position and velocity (x_i, v_i) to a final position (x_f, v_f). From high school physics, we can solve for x and v, given initial values x_i and v_i:

$$x = x_i + v_i t + \frac{1}{2}a_{\max}t^2 \text{ and } v = v_i + a_{\max}t$$

from which, by eliminating t, we have $(v - v_i)^2 = 2a(x - x_i)$. Suppose that the cart starts from rest ($v_i = 0$), accelerates until the midpoint $mid = \frac{1}{2}(x_i + x_f)$, and thereafter decelerates to reach x_f again with velocity $v_f = 0$. The precondition Pre is $x_f > x \wedge v = 0$, the first conjunction saying we move right (we can write another motion primitive for moving left). The assumption A is true, and the guarantee is

$$x_i \leq x \leq x_f \wedge \left(\begin{array}{c} (x_i \leq x \leq mid \Rightarrow v^2 = 2a_{\max}(x - x_i))\wedge \\ (mid \leq x \leq x_f \Rightarrow v^2 = 2a_{\max}(x_f - x)) \end{array} \right)$$

The postcondition is $x = x_f \wedge v = 0$. The footprint provides a bounding box around the cart as it moves, and is given as in Example 3.2. We assume that the motion cannot be interrupted. Thus, we place the annotation $\rightsquigarrow$. The least time to destination is $t_m = 2\sqrt{(x_f - x_i)/a_{\max}}$ and the duration (the interval when it is ready to communicate) is $[t_m, \infty)$. A simpler primitive is m_idle: it keeps the cart stationary. Its assumption is again true, guarantee (and postcondition) is $x = x_i \wedge v = 0$, footprint is a bounding box around the fixed position. It is interruptible by messages from other components (annotation $\lightning$) at any time: $D = [0, \infty)$.

Compatibility. Now consider the constraints that ensure the joint trajectories involving the Cart's motion and the Prod's work primitive are compatible. First, note that the motion of the cart is non-interruptible ($\rightsquigarrow$) but we assume the arm is interruptible, satisfying the constraint in [AGcomp] that at most one motion is not interruptible. Instead of the complexities of modeling the geometry and the dynamics of the arm, we approximate the footprint of the arm (the geometric space it occupies) as a half-sphere centered at the base of the arm and extending upward. Assume first that the motion primitive guarantees that the state is always within this half-sphere. In order for the cart and the robot arm to be compatible, we have to check that the footprints do not overlap. Our guarantee is too weak to prove compatibility, as the footprint of the cart and the arm can intersect when the cart is close to the arm. Instead, we strengthen the guarantee to state that the arm can use the entire sphere when the cart is far and as the cart comes closer the arm effector moves up to make space.

To realise this motion, the cart sends the arm the following information with *arrive*:[1] its current position x_i, its target position x_f, and a lower bound t_{ref} on the time it will take to arrive at x_f. The footprint for the cart's motion can be specified by $\{(\mathsf{x},\mathsf{y},\mathsf{z}) \mid \mathsf{x} \leq x_0 + l/2 + (x_1 - x_0) * t/t_{ref} \wedge \ldots\}$. For the producer, suppose R_{base} is the radius of its base and x_{Prod} its x coordinate. The footprint of the work action is strengthened as $\{(\mathsf{x},\mathsf{y},\mathsf{z}) \mid (\mathsf{z} > \min(ct, h) \vee |\mathsf{x} - x_{\mathsf{Prod}}| \leq R_{base}) \wedge \ldots\}$, where $c \geq h \cdot |x_0 - l - x_{\mathsf{Prod}} - R_{base}|/t_{ref}$. Finally, for compatibility, we need to check that both Cart and Prod can adhere to their footprint and that the footprints are disjoint. Disjointness is satisfied if $x_1 + l < x_{\mathsf{Prod}} - R_{base}$, set this as a precondition to m_move.

Environment Assumptions. We can model a more complex cart moving in a dynamic environment by adding a new component (participant) Env (for *environment*). The physical variables of the environment process encode dynamic properties of the state, such as obstacles in the workspace or external disturbances acting on the cart. We can abstract the environment assumptions into a single motion primitive whose guarantees provide assumptions about the environment behavior to the other components. For example, the environment providing a bounded disturbance of magnitude

[1] For clarity, the type in Figure 2 omits parameters to messages. Our type system uses predicated refinements to reason about parameters.

to the cart's acceleration could be modeled as the guarantee $-1 \leq d \leq 1$ for a physical variable d (for *disturbance*) of Env. The cart can include this input variable in its dynamics: $\dot{v} = u + d$. We can also model sensor and actuator errors in this way. Likewise, we can model obstacles by a physical variable in the environment that denotes the portions of the state space occupied by obstacles, exporting this information through the guarantees, and using this information when the cart plans its trajectory in m_move.

4 MOTION CHOREOGRAPHIES

In this section, we develop a multiparty session typing discipline to prove communication safety and collision freedom. Our session types extend usual multiparty session types by introducing new operators and by reasoning about joint motion in real-time. Moreover, as message exchanges can be a proxy to exchange permissions for motion primitives, our types have predicates as guards in order to model permissions for motion primitives.

4.1 Global Types with Motions and Predicates

We start with a choreography given as a *global type*. The global type constrains the possible sequences of messages and controller actions that may occur in any execution of the system. We extend [Majumdar et al. 2019] by framing and predicate refinements, together with separating conjunctions.

Sorts, ranged over by S, are used to define base types:

$$\mathsf{S} \quad ::= \quad \texttt{unit} \mid \texttt{real} \mid \mathsf{point}(\mathbb{R}^3) \mid \mathsf{vector} \mid \ldots \mid \mathsf{S} \times \mathsf{S}$$

A *predicate refinement* is of the form $\{v : \mathsf{S} \mid \mathcal{P}\}$, where v is a *value variable* not appearing in the program, S a sort, and $\mathcal{P}$ a Boolean predicate. Intuitively, a predicate refinement represents assumptions on the state of the sender and the communicated value to the recipient. We write S as abbreviation for $\{v : \mathsf{S} \mid \mathsf{true}\}$ and $\mathcal{P}$ for $\{v : \mathsf{S} \mid \mathcal{P}\}$ if S is not important.

Definition 4.1 (Global types). *Global types* $(\mathsf{G}, \mathsf{G}', \ldots)$ are generated by the following grammar:

$$\begin{array}{lcl} \mathsf{G} & ::= & g.\mathsf{G} \mid \mathsf{t} \mid \mu\mathsf{t}.\mathsf{G} \\ g & ::= & \mathsf{dt}\langle(\mathsf{p}_i{:}a_i)\rangle \mid \mathsf{p} \to \mathsf{q} : [\mathcal{P}]\ell(\{v : \mathsf{S} \mid \mathcal{P}'\}) \mid g.g \mid g + g \mid g * g \end{array}$$

where p, q range over $\mathbb{P}$, a_i range over abstract motion primitives, and $\mathcal{P}_i, \mathcal{P}'$ range over predicates. g corresponds to the prefix of global types where we do not allow the recursion. We require that $\mathsf{p} \neq \mathsf{q}$, $I \neq \emptyset$, $fv(\mathcal{P}_i) \subseteq \mathsf{p}.X \cup \mathsf{p}.W$, $fv(\mathcal{P}'_i) \subseteq \mathsf{p}.X \cup \mathsf{p}.W \cup \{v\}$, and $\ell_i \neq \ell_j$ 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. We omit the predicate annotation if the predicate is true or not important.

In Definition 4.1, the main syntax follows the standard definition of global types in multiparty session types [Dezani-Ciancaglini et al. 2015; Kouzapas and Yoshida 2013, 2015], with the inclusion of more flexible syntax of choreographies (*separation* $g * g$, *sequencing* $g.g$ and summation $g + g$) and *motion* primitive $\mathsf{dt}\langle(\mathsf{p}_i{:}a_i)\rangle$ extended from [Majumdar et al. 2019], and branching $\mathsf{p} \longrightarrow \mathsf{q}$: $[\mathcal{P}]\ell(\{v : \mathsf{S} \mid \mathcal{P}'\})$.

The motion primitive explicitly declares a *synchronisation* by AG contracts among all the participants p_i. The *branching* type formalises a protocol where participant p first tests if the guard $[\mathcal{P}]$, then sends to q one message with label ℓ and a value satisfying the predicate refinement $\mathcal{P}'$. Recursion is modelled as $\mu\mathsf{t}.\mathsf{G}$, where variable t is bound and guarded in G, e.g., $\mu\mathsf{t}.\mathsf{t}$ is not a valid type. Following the standard session types, in $g_1 + g_2 + \ldots + g_n$, we assume: $g_i = \mathsf{p} \to \mathsf{q} : [\mathcal{P}_i]\ell_i(\{v : \mathsf{S} \mid \mathcal{P}'_i\}).g'_i$ and $\ell_i \neq \ell_j$; and similarly for G. By this rule, hereafter we write: $\mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(\mathcal{P}'{}_i).\mathsf{G}_i\}_{i \in I}$, combining summations and branchings and putting G in the tail into each branching as the standard multiparty session types. $\mathtt{pt}\{\mathsf{G}\}$ denotes a set of participants appeared in G, inductively defined by $\mathtt{pt}\{\mathtt{dt}\langle(\mathtt{p}_1{:}a_1,\ldots,\mathtt{p}_k{:}a_k)\rangle\} = \{\mathtt{p}_1,\ldots,\mathtt{p}_k\}$ and $\mathtt{pt}\{\mathtt{p} \to \mathtt{q} : \ell_i[\mathcal{P}](\{\nu : \mathsf{S} \mid \mathcal{P}'\})\} = \{\mathtt{p},\mathtt{q}\}$ with other homomorphic rules.

A "separating conjunction" $*$ allows us to reason about subsets of participants. It places two constraints on the processes and motion primitives on the two sides of $*$: first, there should not be any communication that crosses the boundary and second, the motion primitives executed on one side should not be coupled (through physical inputs) with motion primitives on the other. We call $g_1 * g_2$ ***fully-separated*** if $\mathtt{pt}\{g_1\} \cap \mathtt{pt}\{g_2\} = \emptyset$, there exist FP_1, FP_2 with $FP_1 \cap FP_2 = \emptyset$, and for every motion primitive $a \vdash (\text{Pre}, A, G, \text{Post}, FP, D, \ddagger)$ in g_i, we have A depends only on state variables in $\mathtt{pt}\{g_i\}$ and for all t, $FP\{t/\mathsf{clock}\} \subseteq FP_i\{t/\mathsf{clock}\}$, for $i \in \{1,2\}$. G is ***fully-separated*** if each $g_1 * g_2$ in G is fully-separated.

Our global type does not include an $\mathtt{end}$ type. An $\mathtt{end}$ introduces an implicit global synchronisation that requires all components to finish exactly at the same time. Informally, "ending" a program is interpreted as robots stopping their motion and staying idle forever (by forever executing an "idle" primitive). Our progress theorem will show that well-typed programs have infinite executions.

Data Flow Analysis on Choreographies. Not every syntactically correct global type is meaningful. We need to check that the AG contracts work together and that the sends and receive work w.r.t the time taken by the different motions. These checks can be performed as a "dataflow analysis" on the tree obtained by unfolding the global type. We start with a few definitions and properties.

Well-scopedness. Consider the unfolding of a global type G as a tree. The leaves of the tree are labeled with motions $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)\rangle$ or message $\mathtt{p} \to \mathtt{q}$, and internal nodes are labeled with the operators $.$, $*$, or $+$. By our assumption, we decorate each $+$ node with the sender and receiver $\mathtt{p} \to \mathtt{q}$ below it.

We say the tree is ***well-scoped*** if there is a way to label the nodes of the tree following the rules below: (1) The root is labeled with the set $\mathbb{P}$ of all participants. (2) If a "." node labeled with $\mathbb{P}'$, both its children are also labeled with $\mathbb{P}'$, if possible. (3) If a "+" node associated with a message exchange $\mathtt{p} \to \mathtt{q}$ is labeled with $\mathbb{P}'$, and both $\mathtt{p},\mathtt{q} \in \mathbb{P}'$, then each of its children are also labeled with $\mathbb{P}'$, if possible. (4) If a "$*$" node is labeled with $\mathbb{P}'$, then we label its children with $\mathbb{P}_1$ and $\mathbb{P}_2$ such that $\mathbb{P}_1 \cap \mathbb{P}_2 = \emptyset$ and $\mathbb{P}_1 \cup \mathbb{P}_2 = \mathbb{P}'$. (5) A leaf node $\mathtt{dt}\langle\mathtt{p}_i : a_i\rangle$ is labeled with $\mathbb{P}'$ if all $\mathtt{p}_i$ are in $\mathbb{P}'$, otherwise the labeling fails. (6) A leaf node $\mathtt{p} \to \mathtt{q}$ is labeled with $\mathbb{P}'$ if both $\mathtt{p},\mathtt{q}$ are in $\mathbb{P}'$, otherwise the labeling fails.

Such a scope labeling is unique if the global type is fully-separated and no participant is "dropped," and so we can define the scope of a node in an unambiguous way.

Unique minimal communication. We first define a notion of *happens-before* with *events* as nodes labeled with motions $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)\rangle$ or message exchanges $\mathtt{p} \to \mathtt{q}$. We define a *happens before* relation on events as the smallest strict partial order such that: there is an edge $n : e \to n' : e'$ if there is an internal node n^* in the tree labeled with . and n is in the left subtree of n^* and n' is in the right subtree of n^*, and one of the following holds: (1) $e \equiv \mathtt{p} \to \mathtt{q}$, $e' \equiv \mathtt{p}' \to \mathtt{q}'$ and $\mathtt{p}'$ is either $\mathtt{p}$ or $\mathtt{q}$; or (2) $\mathtt{pt}\{e\} \cap \mathtt{pt}\{e'\} \neq \emptyset$. We say there is an *immediate happens before* edge $n \to n'$ if $n \to n'$ is in the happens before relation but there is no n'' such that $n \to n''$ and $n'' \to n'$. A global type has ***unique minimal communication*** after motion iff every motion node $n : \mathtt{dt}\langle\mathtt{p}_i : a_i\rangle$ has a *unique* immediate happens before edge to some node $n' : \mathtt{p} \to \mathtt{q}$.

Sender readiness. Consider now a type with unique minimal communication after motion and consider the unique immediate happens before edge $n : \mathtt{dt}\langle(\mathtt{p}_i{:}a_i)\rangle \to n' : (\mathtt{p} \to \mathtt{q})$. Suppose $\mathtt{p}$ is among the processes executing the motion primitives. Since the motion primitives are assumed to be compatible, we know that at most one process is executing a non-interruptible motion, and all the others are executing interruptible motions. Since $\mathtt{p}$ is the next process to send a message, we must ensure that it is the unique process executing the non-interruptible motion. Compatibility ensures that the durations of all other processes are such that they are ready to receive the message from $\mathtt{p}$. We call this condition ***sender readiness***: whenever there is a communication after a motion, the sender of the communication was the unique participant executing a non-interruptible motion; or every process in the motion was executing an interruptible motion. On the other hand, if $\mathtt{p}$ is not among the processes (this can happen when to parallel branches merge), every participant in the motion must have been executing an interruptible motion.

Total synchronisation. It is not enough that $\mathtt{p}$ sends a message to only one other participant: if $\mathtt{p}$ and $\mathtt{q}$ decide to switch to a different motion primitive, every other participant in scope must also be informed. This is ensured by the ***total synchronisation*** between motions: we require that whenever there is a happens before edge between $n : \mathtt{dt}\langle(\mathtt{p}_i : a_i)\rangle$ to $n' : \mathtt{dt}\langle(\mathtt{p}'_i : a'_i)\rangle$, then for every $\mathtt{p}_i$, there is a node where $\mathtt{p}_i$ is a sender or a recipient that happens before n'.

Synchronisability. We call a global type is ***synchronisable*** if it satisfies unique minimal communication, sender readiness, and total synchronisation. Synchronisability of a global type can be checked in time polynomial in the size of the type by unfolding each recursive type once.

Example 4.2 (Synchronisability). We illustrate the synchronisability condition. In general, a node can have multiple outgoing immediate happens before edges. Consider, for distinct participants $\mathtt{p}, \mathtt{q}, \mathtt{p}'$, the type $\mathtt{p} \to \mathtt{q} : \ell.\mathtt{p}' \to \mathtt{q} : \ell'.\mathsf{G}$. The minimal senders are $\mathtt{p}$ and $\mathtt{p}'$, because these two sends cannot be uniquely ordered in an execution. We avoid such a situation because in a process $\mathtt{q}$ we would not know whether to expect a message from $\mathtt{p}$ or from $\mathtt{p}'$ or from both.

Note that synchronisability disallows a sequence of two motions without an intervening message exchange. Thus, $\mathtt{dt}\langle(\mathtt{p}{:}a_1, \mathtt{q}{:}a'_1)\rangle.\mathtt{dt}\langle(\mathtt{p}{:}a_2, \mathtt{q}{:}a'_2)\rangle.\mathsf{G}$ is not well-formed. This is because the implementation of the participants do not have any mechanism to synchronize when to shift from the first motion primitive to the second.

We require the total synchronisation between motions to notify every participant by some message between any change of motion primitives. We disallow the type $\mathtt{dt}\langle(\mathtt{p}{:}a_{11}, \mathtt{q}{:}a_{12}, \mathtt{r}{:}a_{13})\rangle.\mathtt{p} \to \mathtt{q} : \ell.\mathtt{dt}\langle(\mathtt{p}{:}a_{21}, \mathtt{q}{:}a_{22}, \mathtt{r}{:}a_{23})\rangle.\mathsf{G}.$ because $\mathtt{r}$ does not know when to shift from a_{13} to a_{23}.

Note however, that the scoping introduced by $*$ requires messages to be sent only to "local" subgroups. The following type is fine, even though $\mathtt{p}_3$ was not informed when $\mathtt{p}_1$ and $\mathtt{p}_2$ changed motion primitives, as it is in a different branch and there is no happens before edge:

$$\big(\big(\ (\mathtt{dt}\langle(\mathtt{p}_1{:}a_1, \mathtt{p}_2{:}a_2)\rangle.\mathtt{p}_1 \to \mathtt{p}_2 : \ell_1.\mathtt{dt}\langle(\mathtt{p}_1{:}a'_1, \mathtt{p}_2{:}a'_2)\rangle)\ \big) * \mathtt{dt}\langle\mathtt{p}_3{:}a_3\rangle\big)\ .\mathtt{p}_1 \to \mathtt{p}_2{:}\ \ell_1.\mathtt{p}_1 \to \mathtt{p}_3{:}\ \ell_3.G$$

Example 4.3. By inspection, the motion type in Figure 2 is well-scoped and fully separated. It is also synchronizable: to see this, note that for every joint motion primitive, there is at most one motion which is non-interruptible and the participant corresponding to that motion primitive is the unique minimal sender. Moreover, the unique minimal sender sends messages to every participant in the scope of the separating conjunction, thus the type is totally synchronized.

We are ready to define when global types are well-formed.

Definition 4.4 (Well-formed global types). A global type G is *well-formed* if it satisfies the following conditions:

(1) *Total choice:* for each branching type $\mathtt{p} \to \mathtt{q} : \{[\mathcal{P}_i]\ell_i(\mathcal{P'}_i).G_i\}_{i\in I}$, we have $\bigvee_i \mathcal{P}_i$ is valid.
(2) *Well-scoped:* G is well-scoped and fully-separated.
(3) *Total and compatible motion:* For every motion type $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)\rangle$, there exists a motion primitive for each participant in scope and moreover the tuple of motion primitives $(\mathtt{p}_i{:}a_i)$ is compatible.
(4) *Synchronisability:* The global type is synchronisable.

Hereafter we assume global types are well-formed.

Proposition 4.5 (Well-formedness). *Synchronisability is decidable in polynomial time. Checking well-formedness of global types reduces to checking validity in the underlying logic of predicates in global types and motion primitives.*

Proof. All causalities in global types can be checked when unfolding each recursive type once (cf. [Honda et al. 2016, Sec.. 3.5]). Hence synchronisability of a global type can be checked in time polynomial in the size of the type by checking unique minimum communication, sender readiness and total synchronisation simultaneously. By Definition 4.4, checking well-formedness depends on checking validity in the underlying logic. □

4.2 Local Types and Projection

Next, we project global types to their end points against which each end-point process is typed. The syntax of *local types* extends local types from [Majumdar et al. 2019]. Each local type represents a specification for each component.

Definition 4.6 (Local motion session types). The grammar of local types, ranged over by T, is:

$$T ::= \mathsf{dt}\langle a\rangle.T \mid \oplus\{[\mathcal{P}_i]\mathsf{q}!\ell_i(\mathcal{P}'_i).T_i\}_{i\in I} \mid \&\{\mathsf{p}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I} \mid \&\{\mathsf{p}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a\rangle.T \mid \mu\mathsf{t}.T \mid \mathsf{t}$$

where a ranges over motion primitives. 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, types are closed.

Our goal is to *project* a global type onto a participant to get a local type. To define this notion formally, we first need the following definition that merges two global types.

Definition 4.7 (Merging). We define a *merging operator* $\sqcap$, which is a partial operation over global types, as:

$$T_1 \sqcap T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 \\ \&\{\mathsf{p}'?\ell_k(\mathcal{P}_k).T_k\}_{k\in I\cup J} & \text{if } \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I} \text{ and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(\mathcal{P}_j).T_j\}_{j\in J} \end{cases} \\ \&\{\mathsf{p}'?\ell_k(\mathcal{P}_k).T_k\}_{k\in I\cup J}\ \&\ \mathsf{dt}\langle a \sqcap a'\rangle.T' & \text{if } \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a\rangle.T' \text{ and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(\mathcal{P}_j).T_j\}_{j\in J}\ \&\ \mathsf{dt}\langle a'\rangle.T' \end{cases} \\ \&\{\mathsf{p}'?\ell_k(\mathcal{P}_k).T_k\}_{k\in I\cup J}\ \&\ \mathsf{dt}\langle a\rangle.T' & \text{if } \begin{cases} T_1 = \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I} \text{ and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(\mathcal{P}_j).T_j\}_{j\in J}\ \&\ \mathsf{dt}\langle a\rangle.T' \end{cases} \\ \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a \sqcap a'\rangle.T' & \text{if } \begin{cases} T_1 = \mathsf{dt}\langle a\rangle.T' \text{ and} \\ T_2 = \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a'\rangle.T' \end{cases} \\ \&\{\mathsf{p}'?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a\rangle.T' & \text{if } \begin{cases} T_1 = \mathsf{dt}\langle a\rangle.T' \text{ and} \\ T_2 = \&\{\mathsf{p}'?\ell_j(\mathcal{P}_j).T_j\}_{i\in I} \end{cases} \\ T_2 \sqcap T_1 & \text{if } T_2 \sqcap T_1 \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The merge operator for motion $a \sqcap a'$ returns a motion primitive a'' such that $a'' \preceq a$ and $a'' \preceq a'$. We can build such a'' by taking the union of the assumptions and precondition and the intersection of the guarantees, postcondition, and footprint.

Definition 4.8 (Projection). The *projection of a global type onto a participant* r is the largest relation $\upharpoonright_{\mathsf{r}}$ between global and session types such that, whenever $\mathsf{G} \upharpoonright_{\mathsf{r}} T$:

(1) if $\mathsf{G} = \mathsf{p} \to \mathsf{r} : \{[\mathcal{P}_i]\ell_i(\mathcal{P}'_i).\mathsf{G}_i\}_{i\in I}$ then $T = \&\{\mathsf{p}?\ell_i(\mathcal{P}'_i).T_i\}_{i\in I}$ with $\mathsf{G}_i \upharpoonright_{\mathsf{r}} T_i$;
(2) if $\mathsf{G} = \mathsf{r} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(\mathcal{P}'_i).\mathsf{G}_i\}_{i\in I}$ then $T = \oplus\{[\mathcal{P}_i]\mathsf{q}!\ell_i(\mathcal{P}'_i).T_i\}_{i\in I}$ and $\mathsf{G}_i \upharpoonright_{\mathsf{r}} T_i$, $\forall i \in I$;
(3) if $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{[\mathcal{P}_i]\ell_i(\mathcal{P}'_i).\mathsf{G}_i\}_{i\in I}$ and $\mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\}$ then there are T_i, $i \in I$ s.t. $T = \sqcap_{i\in I} T_i$, and $\mathsf{G}_i \upharpoonright_{\mathsf{r}} T_i$, for every $i \in I$;

$$\text{[SUB-MOTION]}\ \frac{a \preceq a' \qquad T \leqslant T'}{\mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a'\rangle.T'} \qquad \text{[SUB-OUT]}\ \frac{\forall i \in I.\ \mathcal{P}_i \Rightarrow \mathcal{P}''_i \land \mathcal{P}'_i \Rightarrow \mathcal{P}'''_i \land T_i \leqslant T'_i}{\oplus\{[\mathcal{P}_i]\mathrm{p}!\ell_i(\mathcal{P}'_i).T_i\}_{i\in I} \leqslant \oplus\{[\mathcal{P}''_i]\mathrm{p}!\ell_i(\mathcal{P}'''_i).T'_i\}_{i\in I\cup J}}$$

$$\text{[SUB-IN1]}\ \frac{\forall i \in I.\ \mathcal{P}'_i \Rightarrow \mathcal{P}_i \land T_i \leqslant T'_i \qquad \mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a'\rangle.T'}{\&\{\mathrm{p}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I\cup J}\ \&\ \mathsf{dt}\langle a\rangle.T \leqslant \&\{\mathrm{p}?\ell_i(\mathcal{P}'_i).T'_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a'\rangle.T'}$$

$$\text{[SUB-IN2]}\ \frac{\forall i \in I.\ \mathcal{P}'_i \Rightarrow \mathcal{P}_i \land T_i \leqslant T'_i}{\&\{\mathrm{p}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I\cup J}\ \&\ \mathsf{dt}\langle a\rangle.T \leqslant \&\{\mathrm{p}?\ell_i(\mathcal{P}'_i).T'_i\}_{i\in I}} \qquad \text{[SUB-IN3]}\ \frac{\mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a'\rangle.T'}{\&\{\mathrm{p}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}\ \&\ \mathsf{dt}\langle a\rangle.T \leqslant \mathsf{dt}\langle a'\rangle.T'}$$

Fig. 5. Subtyping rules

(4) if $\mathsf{G} = \mu\mathrm{t}.\mathsf{G}'$ then $T = \mu\mathrm{t}.T'$ with $\mathsf{G}' \upharpoonright_{\mathrm{r}} T'$ if r occurs in G', otherwise undefined; and
(5) if $\mathsf{G} = g.\mathsf{G}$ then $T = T'.T''$ with $g \upharpoonright_{\mathrm{r}} T'$ and $\mathsf{G} \upharpoonright_{\mathrm{r}} T''$.[2]
(6) $\mathsf{dt}\langle(\mathrm{p}_i{:}a_i)\rangle \upharpoonright_{\mathrm{r}} \mathsf{dt}\langle a_j\rangle$ with $\mathrm{r} = \mathrm{p}_j$;
(7) $(g_1 * g_2) \upharpoonright_{\mathrm{r}} T_i$ and $g_i \upharpoonright_{\mathrm{r}} T_i$ if $\mathrm{r} \in \mathrm{pt}\{g_i\}$ $(i \in \{1,2\})$
(8) $g_1.g_2 \upharpoonright_{\mathrm{r}} T_1.T_2$ with $g_i \upharpoonright_{\mathrm{r}} T_i$ $(i = 1,2)$

We omit the cases for recursions and selections (defined as [Scalas and Yoshida 2019, Section 3]). The branching prefix is defined as the branching in (1-3).

Example 4.9 (Projection of Figure 2). For the example from Sec. 2, the local type of the cart is:

$$\mu\mathrm{t}.\left(\begin{array}{l} (\mathsf{GRobot}!\mathit{arrive}(t).\mathsf{RRobot}!\mathit{free}. \\ \ \mathsf{dt}\langle \mathsf{m\_move}(\mathsf{GRobot})\rangle.\mathsf{GRobot}!\mathit{ready}.\mathsf{dt}\langle \mathsf{m\_idle}\rangle.\mathsf{GRobot}?\mathit{ok}.\mathsf{dt}\langle \mathsf{m\_move}\rangle) \\ \&\ (\mathsf{RRobot}!\mathit{arrive}(t).\mathsf{GRobot}!\mathit{free}.\ldots\text{symmetric}\ldots) \end{array}\right).\mathrm{t}$$

4.3 Subtyping and Typing Processes

The local types are a specification for the processes and there is some freedom to implement these specifications. The subtyping relation helps bridge the gap between the specification and the implementation.

Definition 4.10 (Subtyping). Subtyping $\leqslant$ is the largest relation between session types coinductively defined by rules in Figure 5.

A subtype has fewer requirements and provide stronger guarantees. The subtyping of motion primitives ([SUB-MOTION]) allows replacing an abstract motion primitive by a concrete one if a refines a' and $T \leqslant T'$. For internal choice and sending ([SUB-OUT]), the subtyping ensure the all the messages along with the associated predicate are allowed by the supertype. Predicate refinements are converted to logical implication.

For the external choice and message reception ([SUB-IN1,2,3]), subtyping makes sure the process reacts properly to the messages expected by the super type. A subtype can have cases for more messages but they don't matter. The subtype guarantees that the process will never receive theses messages. On the other hand, the sender of the messages has to be the same. We enforce this directly in the syntax of the programming language and the type system. It may seem that ([SUB-IN3]) introduces a new sender in the subtype but this rule is correct because, in our synchronous model,

[2]We abuse T to denote *a local type prefix* which is given replacing t and $\mu\mathrm{t}.T$ by ϵ (as defined for global type prefix g).

Table 1. Typing rules for motion processes.

$$\text{[T-SUB]}\ \frac{\Gamma,\Sigma \vdash P : T \quad T \leqslant T'}{\Gamma,\Sigma \vdash P : T'} \qquad \text{[T-REC]}\ \frac{\Gamma \cup \{\mathtt{x} : T\}, \mathsf{true} \vdash P : T}{\Gamma,\Sigma \vdash \mu\mathtt{x}.P : T} \qquad \text{[T-MOTION]}\ \frac{\Gamma, a.\mathrm{Post} \vdash Q : T \qquad \Sigma \Rightarrow a.\mathrm{Pre}}{\Gamma,\Sigma \vdash \mathsf{dt}\langle a\rangle.Q : \mathsf{dt}\langle a\rangle.T}$$

$$\text{[T-OUT]}\ \frac{\Sigma \Rightarrow \mathcal{P} \wedge \mathcal{P}'\{e/v\} \qquad \Gamma,\Sigma \vdash \mathsf{e} : S \qquad \Gamma,\Sigma \vdash P : T}{\Gamma,\Sigma \vdash \mathtt{q}!\ell(\mathsf{e}).P : [\mathcal{P}]\mathtt{q}!\ell(\{v : S \mid \mathcal{P}'\}).T} \qquad \text{[T-CHOICE1]}\ \frac{\forall i \in I \quad \Gamma \cup \{x_i : S_i\}, \Sigma \wedge \mathcal{P}_i\{x_i/v\} \vdash P_i : T_i}{\Gamma,\Sigma \vdash \sum_{i \in I} \mathtt{q}?\ell_i(x_i).P_i : \&\{\mathtt{q}?\ell_i(\mathcal{P}_i).T_i\}_{i\in I}}$$

$$\text{[T-CHOICE2]}\ \frac{\forall i \in I \quad \Gamma \cup \{x_i : S_i\}, \Sigma \wedge \mathcal{P}_i\{x_i/v\} \vdash P_i : T_i \quad \Gamma,\Sigma \vdash \mathsf{dt}\langle a\rangle.Q : T}{\Gamma,\Sigma \vdash \sum_{i\in I} \mathtt{q}?\ell_i(x_i).P_i + \mathsf{dt}\langle a\rangle.Q : \&\{\mathtt{q}?\ell_i(\{v : S \mid \mathcal{P}_i\}).T_i\}_{i\in I} \,\&\, T}$$

$$\text{[T-COND]}\ \frac{\Gamma \vdash \mathsf{e} : \mathsf{bool}\ \exists k \in I\ \Sigma \wedge \mathsf{e} \Rightarrow \mathcal{P}_k\ \Gamma,\Sigma \wedge \mathsf{e} \vdash P_1 : T_k\ \Gamma,\Sigma \wedge \neg\mathsf{e} \vdash P_2 : \oplus\{[\mathcal{P}_i]T_i\}_{i\in I\setminus\{k\}}}{\Gamma,\Sigma \vdash \mathsf{if\ e\ then}\ P_1\ \mathsf{else}\ P_2 : \oplus\{[\mathcal{P}_i]T_i\}_{i\in I}}$$

$$\text{[T-SESS]}\ \frac{\forall i \in I \quad \emptyset, \mathcal{P}_i \vdash P_i : \mathsf{G}\!\upharpoonright\!\mathtt{p}_i \qquad \mathsf{pt}\{\mathsf{G}\} = \mathbb{P} \qquad \forall j \in I.\ j \neq i \Rightarrow \mathsf{geom}\left(\mathtt{p}_i\right)(\mathcal{P}_i) \cap \mathsf{geom}\left(\mathtt{p}_j\right)(\mathcal{P}_j) = \emptyset}{\bigwedge_{i\in I} \mathcal{P}_i \vdash \prod_{i\in I} \mathtt{p}_i \triangleleft P_i : \mathsf{G}}$$

sending is blocking and cannot be delayed. Therefore, if the supertype only contains a motion, we know for a fact that no messages can arrive unexpectedly in the subtype.

Our subtyping conditions generalise those of motion session types of Majumdar et al. [2019] in multiple ways. First, we allow refinement of motion primitives ([SUB-MOTION]) as a way to abstract trajectories. In [Majumdar et al. 2019], the actions are abstract symbols and are fixed statically. Second, the rules for communication must check predicate refinements—in [Majumdar et al. 2019], global types did not have refinements.

The final step in our workflow checks that the process of each participant in a program implements its local type using the typing rules from Table 1. Our typing rules additionally maintain a logical context to deal with the predicate refinements [Rondon et al. 2008].

We write $\Gamma, \Sigma \vdash S : T$ to indicate the statement S has type T under the variable context Γ and the logical context Σ. Γ is the usual typing context; Σ is a formula characterising what is known about the state of the system when a process is about to execute. The typing rules are shown in Table 1. [T-MOTION] considers Σ derives the pre-condition of motion a, while Q guarantees its post-condition. [T-OUT] assumes Σ of P derives a conjunction of the guard and refined predicates declared in types. [T-CHOICE1] is its dual and [T-CHOICE2] includes the default motion branch. [T-COND] is similar with the usual conditional proof rule. [T-SESS] combines well-typed participants each of which follows a projection of some global type G given assumptions over the initial state $\bigwedge_{i\in I} \mathcal{P}_i$ of each participant. We requires the initial state to be collision free as the global types only maintains collision freedom.

4.4 Soundness

The soundness of multiparty sessions is shown, using subject reduction (typed sessions reduce to typed sessions) and progress. In order to state soundness, we need to formalise how global types are reduced when local session types evolve. To define the consumption of global types, as done for processes in Sec. 3.3, we extend the syntax of global types to allow *partial consumption* of motion types. In addition, we extend g, annotating each motion type with its unique minimal sender, $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@t\rangle^{\mathtt{p}}$ for $t \geq 0$.

Definition 4.11 (Global types consumption and reduction). The *consumption* of the communication $\mathtt{p} \xrightarrow{\ell} \mathtt{q}$ or motion $\mathtt{dt}\langle a\rangle[t_1, t_2]$ for the global type G (notation $\mathsf{G} \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q}$ and $\mathsf{G} \setminus \mathtt{dt}\langle a\rangle[t_1, t_2]$) is the global type defined (up to unfolding of recursive types) using the following rules:

(1) $(\mathtt{p} \to \mathtt{q} : \{\ell_i.\mathsf{G}_i\}_{i\in I}) \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q} = \mathsf{G}_k$ if there exists $k \in I$ with $\ell = \ell_k$;
(2) $(\mathtt{r} \to \mathtt{s} : \{\ell_i.\mathsf{G}_i\}_{i\in I}) \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q} = \mathtt{r} \to \mathtt{s} : \{\ell_i.(\mathsf{G}_i \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q})\}_{i\in I}$ if $\{\mathtt{r}, \mathtt{s}\} \cap \{\mathtt{p}, \mathtt{q}\} = \emptyset$;
(3) $(g_1 * g_2) \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q} = (g_1' * g_2)$ if $\{\mathtt{p}, \mathtt{q}\} \subseteq \mathtt{pt}\{g_1\}$ and $g_1 \setminus \mathtt{p} \xrightarrow{\ell} \mathtt{q} = g_1'$ and symmetrically if $\{\mathtt{p}, \mathtt{q}\} \subseteq \mathtt{pt}\{g_2\}$;
(4) $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)\rangle \setminus \mathtt{dt}\langle(\mathtt{p}_i{:}a_i[t_i; t_i'])\rangle = \mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@0\rangle^{\mathtt{p}}$ where $\mathtt{p}$ is the unique minimal sender in G.
(5) $\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@t\rangle^{\mathtt{p}} \setminus \mathtt{dt}\langle(\mathtt{p}_i{:}a_i[t_i; t_i'])\rangle = \mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@t'\rangle^{\mathtt{p}}$ if $t' > t$ and for all i, $t_i' - t_i = t' - t$ and $t' \leq \lceil a_i.D\rceil$ where $\lceil D\rceil$ denotes the upper bound of the interval D;
(6) $(g_1 * g_2) \setminus \mathtt{dt}\langle(\mathtt{p}_i{:}a_i[t_i; t_i'])_{\mathtt{p}_i\in\mathtt{pt}\{g_1\}\uplus\mathtt{pt}\{g_2\}}\rangle = (g_1' * g_2')$ if $g_j \setminus \mathtt{dt}\langle(\mathtt{p}_i : a_i[t_i; t_i'])_{\mathtt{p}_i\in\mathtt{pt}\{g_j\}}\rangle = g_j'$ $(j = 1, 2)$
(7) $g \setminus x = g'$ then $g.g_1 \setminus x = g'.g_1$ and $g.\mathsf{G} \setminus x = g'.\mathsf{G}$.
(8) $g \setminus x = g'$ if $\exists g''.\exists\vec{\mathtt{p}}.\ \emptyset \rhd g \rightsquigarrow^* \vec{\mathtt{p}} \rhd g''$ and $g'' \setminus x = g'$ where C is a reduction context of the prefix defined as: $C = C.g \mid C * g \mid g * C \mid [\ ]$ and $\vec{\mathtt{p}}$ is a set of enabled senders such that $\vec{\mathtt{p}} \subseteq \mathbb{P}$ and
   (a) $\vec{\mathtt{p}} \rhd C[\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@t\rangle^{\mathtt{p}}] \rightsquigarrow \vec{\mathtt{p}} \cup \{\mathtt{p}\} \rhd C$ if $\exists i.\ \mathtt{p}_i = \mathtt{p} \wedge t \in D_i \wedge \ddagger_i = \rightsquigarrow$.
   (b) $\vec{\mathtt{p}} \rhd C[\mathtt{dt}\langle(\mathtt{p}_i{:}a_i)@t\rangle^{\mathtt{p}}] \rightsquigarrow \vec{\mathtt{p}} \rhd C$ with $\mathtt{p} \in \vec{\mathtt{p}}$ and for all i, $\mathtt{p}_i \neq \mathtt{p}$ and $t \in D_i$.

In (6) and (7), we write x as either $\mathtt{p} \to \mathtt{q}$ or $\mathtt{dt}\langle a\rangle[t_1, t_2]$. The *reduction of global types* is the smallest pre-order relation closed under the rule: $\mathsf{G} \Longrightarrow \mathsf{G}\setminus\mathtt{p} \xrightarrow{\ell} \mathtt{q}$ and $\mathsf{G} \Longrightarrow \mathsf{G}\setminus\mathtt{dt}\langle\mathtt{p}_i{:}a_i[t_i, t_i']\rangle$.

Note that the above reduction preserves well-formedness of global types. We can now state the main results. Our typing system is based on one in [Ghilezan et al. 2019b] with motion primitives and refinements. Since the global type reduction in Definition 4.11 is only related to communications or synchronisation of motions but not refinements, (1) when the two processes $\mathtt{p}$ and $\mathtt{q}$ synchronise by a communication, its global type can always reduce; or (2) when all processes synchronise by the same motion action, their corresponding global type can always be consumed.

Thus process behaviours correspond to reductions of global types, which is formulated as in the following theorem. Recall $\xrightarrow{\alpha}$ denotes any transition relation or reduction.

Theorem 4.12 (Subject Reduction). *Let M be a multiparty session, G be a well-formed global type, and a physical state I such that $I \vdash M : \mathsf{G}$. For all M', if $M \xrightarrow{\alpha} M'$, then $I' \vdash M' : \mathsf{G}'$ for some G', I' such that $\mathsf{G} \Longrightarrow \mathsf{G}'$. Thus, if $M \equiv M'$ or $M \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} M'$, then $\vdash M' : \mathsf{G}'$ for some G' such that $\mathsf{G} \Longrightarrow \mathsf{G}'$.*

See [Majumdar et al. 2020] for the detailed proofs.

Below we state the two progress properties as already explained in Sec. 2. The first progress is related to communications, while the second one gurantees the typed multiparty session processes are always collision free. As a consequence, if $I \vdash M : \mathsf{G}$ for a well-formed type, then M does not get stuck and can always reduce.

Theorem 4.13 (Progress). *Let M be a multiparty session, G be a well-formed type, and a physical state I. If $I \vdash M : \mathsf{G}$ then:*

(1) (Communication and Motion Progress) *there is M' such that $M \longrightarrow M'$ and*
(2) (Collision-free Progress) *If $M \longrightarrow^* M'$, then M' is also collision free.*

$$\begin{array}{rl} & \mathsf{Cart} \to \mathsf{Prod} : \mathit{arrive}.\mathsf{Cart} \to \mathsf{GRobot} : \mathit{start}.\mathsf{Cart} \to \mathsf{RRobot} : \mathit{start}. \\ \text{Initial Phase} \ \equiv & \left(\begin{array}{l} \left(\begin{array}{l} (\mathsf{dt}\langle\langle \mathsf{Cart} : \mathsf{m\_move}^{\rightsquigarrow}(\mathsf{Prod}), \mathsf{Prod} : \mathsf{work}^{\lightning} \rangle\rangle^{\mathsf{Cart}}. \\ \mathsf{Cart} \to \mathsf{Prod} : \mathit{ready}.\mathsf{dt}\langle \mathsf{Cart} : \mathsf{m\_idle}^{\lightning}, \mathsf{Prod} : \mathsf{place}^{\rightsquigarrow} \rangle)^{\mathsf{Prod}}. \end{array} \right) \\ \qquad * \ \mathsf{dt}\langle \mathsf{RRobot} : \mathsf{work}^{\lightning} \rangle^{\mathsf{Prod}} \ * \ \mathsf{dt}\langle \mathsf{GRobot} : \mathsf{work}^{\lightning} \rangle^{\mathsf{Prod}} \end{array} \right) \end{array}$$

$$\text{Process Green Item} \ \equiv$$

$$\left(\mathsf{dt}\langle \mathsf{Prod} : \mathsf{work}^{\lightning} \rangle^{\mathsf{Cart}}) * \left(\begin{array}{l} \mathsf{Cart} \to \mathsf{GRobot} : \mathit{arrive}.\mathsf{Cart} \to \mathsf{RRobot} : \mathit{free}. \\ \left(\text{as in the green box in Figure 2} \quad * \ \mathsf{dt}\langle \mathsf{RRobot} : \mathsf{work}^{\lightning} \rangle^{\mathsf{Cart}} \right) \end{array} \right) \right)$$

$$\text{Global Type} \ \equiv \ \mu \mathrm{t}.\langle \text{Initial Phase} \rangle. \left(\begin{array}{l} (\mathsf{Prod} \to \mathsf{Cart} : \mathit{green}.\langle \text{Process Green Item} \rangle) \\ +(\mathsf{Prod} \to \mathsf{Cart} : \mathit{red}.\langle \text{Process Red Item} \rangle) \end{array} \right).\mathrm{t}$$

Fig. 6. Motion session type annotated with minimal senders. For readability, we have broken the type into sub-parts: the initial phase, and processing items (the type for processing red items is symmetric and omitted)

Proof. (*Outline – see [Majumdar et al. 2020] for the detailed proofs.*)

(Communication Progress) The proof is divided into two cases: (a) the guard appearing in the branching type followed by (b) a message exchange between two parties. Case (a) follows from the fact that there is always at least one guard in a choice whose predicate is evaluated to true (Definition 4.4(1)). The typing rules ensure that the predicates are satisfied when a message is sent. Case (b) is proved using Theorem 4.12 with Definition 4.4(4) since the unique minimal sender in G can always send a message.

(Motion Progress) For executing motions, Definition 4.4(3) and Theorem 3.5 ensure local trajectories can be composed into global trajectories.

(Collision-free Progress) By induction. Under I and by [T-SESS], M is initially collision free. For the communication, note that it does not change the physical state, and therefore, does not impact the geometric footprint used by a process. Hence the case $\mathsf{G} \setminus \mathsf{p} \xrightarrow{\ell} \mathsf{q}$ is straightforward. For motion actions, Definition 4.4(3) and Theorem 3.5 ensures collision freedom through execution steps. For a collision free program, collision freedom of the next state follows from compatibility of motion. □

4.5 A More Complex Coordination Example with Producer

We now discuss an extended version of the coordination example from Sec. 2 that we shall implement on physical robots in Sec. 5. In addition to cart and two arms, we add a producer robot. The Prod generates green or red parts and places them on to the cart. The cart Cart carries the part to the two robot arms as before. We shall use this extended version as the basis for one of our case studies.

The coordination protocol is as follows. First, the protocol starts by syncing all participants. Then, the cart moves to the producer after announcing a message *arrive*. While the cart moves to the producer, the two consumer robots can work independently. When the cart is at the producer, it synchronises through a message *ready*, and idles while the producer places an object on to it. When the producer is done, it synchronises with the cart, and tells it whether the object is green or red. Based on this information, the cart tells one of the consumers that it is arriving with a part and tells the other consumer that it is free to work. Subsequently, the cart moves to the appropriate consumer to deposit the part, while the other consumer as well as the producer is free to continue their work. When the robot is at the consumer, it syncs through a message, and idles until the part is taken off. After this, the protocol starts again as the cart makes its journey back to the producer, while the consumers independently continue their work.

Global Type. The global type for the example extends one in Figure 2, and is shown in Figure 6. It can be seen that the global type has total choice (trivially), and is well-scoped and synchronisable. The motion primitive specifications are omitted; we ensure in our evaluation that the motion primitives are compatible and the type is fully separated using calls to the SMT solver dReal.

Processes. The processes in this example extend the processes from Sec. 2 but the RRobot and GRobot processes are as before:

$$\begin{array}{ll}
\mathsf{Cart}: & \mu \mathtt{x}.\mathsf{Prod}!\mathit{arrive}.\mathsf{RRobot}!\mathit{start}.\mathsf{GRobot}!\mathit{start}.\mathsf{m\_move}(\text{co-ord of }\mathsf{Prod}).\mathsf{Prod}!\mathit{ready}.\mathsf{m\_idle}. \\
& \;(\mathsf{Prod}?\mathit{red}.\mathsf{RRobot}!\mathit{arrive}.\mathsf{GRobot}!\mathit{free}.\mathsf{m\_move}(\text{co-ord of }\mathsf{RRobot}). \\
& \qquad \mathsf{RRobot}!\mathit{ready}.\mathsf{m\_idle}.\mathsf{RRobot}?\mathit{ok}.\mathtt{x} \\
& \;+\mathsf{Prod}?\mathit{green}.\text{ symmetrically for }\mathsf{GRobot}\;) \\
\mathsf{Prod}: & \mu \mathtt{x}.\mathsf{Cart}?\mathit{arrive}.\mathsf{work}.\mathsf{Cart}?\mathit{ready}.\mathsf{place}.(\mathsf{Cart}!\mathit{green} + \mathsf{Cart}!\mathit{red}).\mathsf{work}.\mathtt{x}
\end{array}$$

Local Types. The local types for the components are:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{Cart}:} \\
\mu \mathrm{t}. & \mathsf{Prod}!\mathit{arrive}(t).\mathsf{RRobot}!\mathit{start}.\mathsf{GRobot}!\mathit{start}. \\
& \mathsf{dt}\langle \mathsf{m\_move}(\mathsf{Prod})\rangle.\mathsf{Prod}!\mathit{ready}.\mathsf{dt}\langle \mathsf{m\_idle}\rangle. \\
& (\,(\mathsf{Prod}?\mathit{green}.\mathsf{GRobot}!\mathit{arrive}(t).\mathsf{RRobot}!\mathit{free}. \\
& \;\mathsf{dt}\langle \mathsf{m\_move}(\mathsf{GRobot})\rangle.\mathsf{GRobot}!\mathit{ready}. \\
& \;\mathsf{dt}\langle \mathsf{m\_idle}\rangle.\mathsf{GRobot}?\mathit{ok}) \\
& \&\ (\mathsf{Prod}?\mathit{red}.\ldots\text{symmetric}\ldots)\,).\mathrm{t}
\end{array}$$

$$\begin{array}{l}
\mathsf{Prod}: \\
\quad \mu \mathrm{t}.\mathsf{Cart}?\mathit{arrive}(t).\mathsf{dt}\langle \mathsf{work}\rangle. \\
\qquad \mathsf{Cart}?\mathit{ready}.\mathsf{dt}\langle \mathsf{place}\rangle. \\
\qquad (\mathsf{Cart}!\mathit{green}.\mathsf{dt}\langle \mathsf{work}\rangle.\mathrm{t} \\
\qquad \;\oplus\; \mathsf{Cart}!\mathit{red}.\mathsf{dt}\langle \mathsf{work}\rangle.\mathrm{t}) \\
\mathsf{RRobot}, \mathsf{GRobot}: \\
\quad \mu \mathrm{t}.(\mathsf{Cart}?\mathit{start}.\mathsf{dt}\langle \mathsf{work}\rangle. \\
\qquad (\,(\mathsf{Cart}?\mathit{arrive}(t).\mathsf{dt}\langle \mathsf{work}\rangle. \\
\qquad \mathsf{Cart}?\mathit{ready}.\mathsf{dt}\langle \mathsf{pick}\rangle.\mathsf{Cart}!\mathit{ok}) \\
\qquad \&\ (\mathsf{Cart}?\mathit{free}.\mathsf{dt}\langle \mathsf{work}\rangle)\,).\mathrm{t}
\end{array}$$

We can show that all the processes type check. From the soundness theorems, we conclude that the example satisfies communication safety, motion compatibility, and collision freedom.

5 IMPLEMENTATION AND CASE STUDY

Implementation. Our implementation has two parts. The first part takes a program, a specification for each motion primitive, and a global type and checks that the type is well-formed and that each process satisfies its local type. The second part implements the program on top of the Robotic Operating System (ROS) [Quigley et al. 2009], a software ecosystem for robots. We reuse the infrastructure of motion session types [Majumdar et al. 2019]; for example, we write programs in PGCD syntax [Banusic et al. 2019]. The verification infrastructure is about 4000 lines of Python code, excluding the solver. The code is available at https://github.com/MPI-SWS/pgcd and instructions to run these experiments are located in the `oopsla20_artifact` branch.

Internally, we represent programs and global types as state machines [Deniélou and Yoshida 2012], and implement the dataflow analysis on this representation. Additionally, we specify motion primitives in the local co-ordinate system for each robot and automatically perform frame transformations between two robots. The core of ROS is a publish-subscribe messaging system; we extend the messaging layer to implement a synchronous message-passing layer. Our specifications contain predicates with nonlinear arithmetic, for example, to represent footprints of components as spheres. Obstacle are represented as passive components, i.e., components which have a physical footprint but do not execute program. Such components can interact with normal components through input and state variables and their are considered when checking the absence of collision. On the other hand, the obstacles are excluded from the checks related to communication, e.g., synchronisability. We use the dReal4 SMT solver [Gao et al. 2013] to discharge validity queries. The running times are obtained on a Intel i7-7700K CPU at 4.2GHz and dReal4 running in parallel on 6 cores.

Tests. We evaluate our system on two benchmarks and a more complex case study: (1) We test scalability of the verification using micro-benchmarks. (2) We compare our approach with previous approaches on a set of robotic coordination scenarios from the literature [Banusic et al. 2019; Majumdar et al. 2019], and (3) As a large case study, we verify and implement a complex choreography example based on a variation of the example from Sec. 4.5.

(1) Micro-benchmarks. The micro-benchmarks comprise a parametric family of examples that highlight the advantage of our motion session calculus, specifically the separating conjunction, over previous typing approaches [Majumdar et al. 2019]. The scenario consists of carts moving back and forth along parallel trajectories, and is parameterised by the number of carts. Fig. 7 shows the verification times for this example in the two systems. We start with 2 robots and increase the number of robots until we reach a 200 s. timeout for the verification. The previous calculus is discrete-time and does not separate independent components. Thus, the type system synchronises every process and, therefore, generates complex collision checks which quickly overwhelms the verifier. Our global types use the separating conjunction to split the specifications into independent pieces and the verifcation time increases linearly in the number of processes.

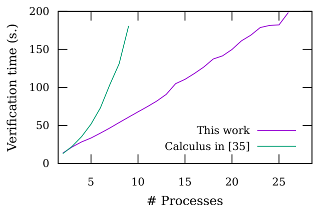


Fig. 7. Parallel lanes micro-benchmarks

Setup for (2) and (3). For examples used previously in related works [Banusic et al. 2019; Majumdar et al. 2019], we write specifications using our motion session types, taking advantage of the modularity of the type system. For the new case study (part (3)), we implement a variation of the example presented in Sec. 2, where we decouple the producer placing an object and a sensor that determines if the object is green or red. Thus, after the producer places an object on the cart, the cart first moves to the sensor, communicates with the sensor to get the color, and then delivers the object as described in the protocol. We filmed our experiments and a short video can be seen in the ***supplementary materials***. We use three robots: a custom-built robot arm modified from an open-source arm, a commercial manipulator, and a mobile cart, as shown in Figure 8b. We use placeholders for the producer arm and color sensor, as we do not have additional hardware (and arms are expensive). The robots are built with a mix of off-the-self parts and 3D printed parts as described below.

Arm. The arm is a modified BCN3D MOVEO (https://github.com/BCN3D/BCN3D-Moveo). The upper arm section is shortened to make it lighter and easier to mount on a mobile cart. It has three degrees of freedom. The motion primitives consist of moving between poses and opening/closing the gripper. The motion is a straight line in the configuration space (angles of the joints) which corresponds to curves in physical space.

Panda Arm. The Panda arm by Franka Emika (https://www.franka.de/technology) is a seven degrees of freedom commercial manipulator platform. It is controlled similarly to the MOVEO arm but with a more complex configuration space. The Panda arm has a closed-loop controller and can get to a pose with an error less than 0.1mm. The controller also comes with collision detection using feedback from the motors. We command this arm using a library of motion primitives provided by the manufacturer.

Mobile Carts. There are two carts. Both are omnidirectional driving platforms. One uses mecanum wheels and the other 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. Its basic motion primitives are moving in the direction of the wheels, moving perpendicularly to the direction of the wheels, and rotating around its center.

The arm and cart are equipped with RaspberryPi 3 model B to run their processes and use stepper motors. This enables a control of the joints and wheels with less than 1.8 degree of error. However, these robots do not have global feedback on their position and keep track of their state using *dead reckoning*. This can be a challenge for the cart, which keeps accummulating error over time. As our example is a loop, we manually reset the cart's position when it gets back to the producer after delivering the object. (A more realistic implementation would use feedback control, but we omit this because control algorithms are a somewhat orthogonal concern.) For the producer and the color sorter, we run the processes but have placeholders in the physical world and realise the corresponding action manually. All computers run Ubuntu 18.04 with ROS 2 Dashing Diademata.

Table 3 shows the specification for the robots and their motions. As the two carts share most of their specification, we group them together. The specification includes the geometrical description of the robots and the motion primitives.

(2) Revisiting the Examples from PGCD. As we build on top of PGCD, we can use the examples used to evaluate that system. We take 4 examples, for which we compare the specification effort in the previous calculus [Majumdar et al. 2019] to our new calculus. As the two calculi have different models for the time and synchronisation, we made some minor adaptations such that the same programs can be described by the two calculi. The old calculus requires motion primitives to have fixed duration and does not support interruptible motions. Furthermore, there is no "$*$" in the old calculus so all the motions are always synchronised. In our new specification, we take advantage of our richer calculus to better decompose the protocol. We use the following scenarios.

Fetch. in this experiment, the Moveo arm is attached on top of a cart. The goal is to get an object. The cart moves toward the object until the object if within the arm's reach. The arm grabs the object and the cart goes back to its initial position.

Handover. This experiment is a variation of the previous one. There are two carts instead of one and the object to fetch is on top of the new cart. The two carts meet before the arm takes an object placed on top of the second cart and then, both go back to their initial positions.

Twist and Turn. In this experiment, the two carts start in front of the each other. The arm takes an object from the small cart. Then all three robots move simultaneously.
The cart carrying the arm rotates in place, the other cart describes a curve around the first cart, and arm moves from one side of the cart to the other side. At the end, the arm puts the object on the carrier.

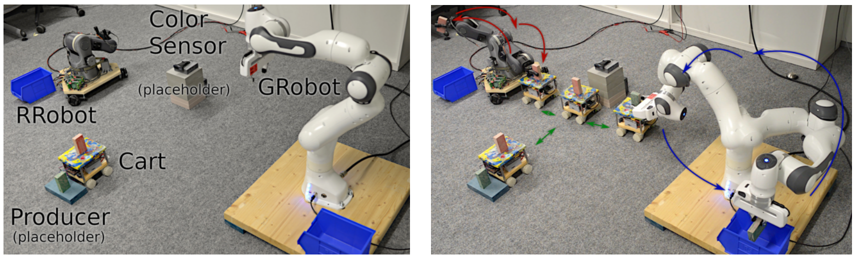
Underpass. In this experiment, the arm and the cart cooperate to go under an obstacle. First, the cart goes toward the arm, which takes the object from the cart. The cart goes around the arm passing under an obstacle. Finally, the arm puts the object back on the cart on the other side of the obstacle.

Table 2 reports the size of the examples and their specifications, as well as the verification time. The verification time is broken down into syntactic well-formedness checks on the choreography, the motion compatibility checks for the trajectories, and typing the processes w.r.t. local types. The motion compatibility checks dominate the verification time. When we compare our new calculus to the previous approach, we can observe that the global types are slightly larger but the verification time can be significantly smaller. The increase in specification size comes with the addition of new Assume-Guarantee contracts when $*$ is used.

The verification time from the existing specification are higher than the time reported in the previously published results. When implementing our new calculus, we found and fixed some bugs

Table 2. Programs, specification, and verification time for the PGCD examples. “Prev” refers to [Majumdar et al. 2019].

Scenario	Program (Loc)			Global Type (LoC)		Syntactic Checks (s.)		Motion Compatibility (s.)		Typing (s.)	
	Arm	Cart 1	Cart 2	Prev	This work	Prev	This work	Prev	This work	Prev	This work
Fetch	14	19	–	22	42	0.1	0.1	> 3600 [3]	7	0.1	0.1
Handover	9	8	6	14	26	0.1	0.1	2224	70	0.1	0.1
Twist and turn	12	16	6	15	17	0.1	0.1	599	203	0.1	0.1
Underpass	18	3	14	21	65	0.1	0.1	187	114	0.1	0.1



(a) Setup (b) Sequence of motions

Fig. 8. Experimental setup for the sorting example

in the verifier code from PGCD. Those bugs resulted in incomplete collision checks and fixing them increased the burden on the SMT solver.[3]

(3) A Complex Case Study. The purpose of the case study is to show that we can semi-automatically verify systems beyond the scope of previous work. Table 3 shows the sizes of the processes (in lines of code, in the syntax of PGCD) and statistics related to the global specification and verification time. The global specification consists of the global type, the environment description, and (manually provided) annotations for the verification. The annotations are mostly related to the footprints and the parallel composition. Each time we use the parallel composition operator we specify a partition of the current footprint (the “$\exists FP_1, FP_2$” in rule [AGcomp]).

In Table 3, we split the running times into the well-formedness checks which can be done syntactically, the verification conditions for the execution of the motion primitives, and the typing. We did not try to encode this example using the older PGCD specifications for two reasons. First, the stronger requirement on discrete time steps and global synchronisation in time in the previous calculus would subtantially change this example. Remember that motions in [Majumdar et al. 2019] are specified simultaneously for all the robots and must have the same duration. When multiple motion primitives have different durations, a motion step can only be as long as the shortest motion. Thus, we would need to chunk the longer motions into a sequence of smaller steps. This would, in turn, make the (sub)typing much more difficult as one motion step in the code would correspond to a sequence of motion steps in the specification. Second, as the previous calculus does not include the separating conjunction, there is little hope that the SMT solver can cope with the compelxity of this example: it contains more robots and more complex ones. Because our global types include separating conjunctions, we can more efficiently generate and discharge the verification conditions. At the modest cost of specifying the footprints for each thread, the collision checks only need to

[3] Two out of 217 queries in the Fetch example are particularly hard for the solver and could not be solved with 1 hour. We suspect a bug in the solver as all the other queries are solved in 134s.

Table 3. Programs, motion specification, type, and verification time

Robot	Motion Spec (LoC)	Program (LoC)	Specification		
Moveo arm	306	28	Global type	96	LoC
Panda arm	348	31	Syntactic checks	0.2	s.
Carts	404	45	Compatibility checks	2410	s.
Sensor/Producer	105	17 / 10	Typing	3.1	s.

consider the robots within a thread and not all the robots in the system. This brings a substantial reduction in the complexity of the verification conditions.

In conclusion, the case study demonstrates the power of our method in specifying and verifying non-trivial coordination tasks between multiple robots. It requires the expressiveness of our motion primitive specifications and the modularity of our separating conjunction.

6 RELATED WORK

The field of concurrent robotics has made enormous progress—from robot soccer to self-driving systems and to industrial manufacturing. However, to the best of our knowledge, none of these impressive systems come with formal guarantees of correctness. Our motivation, like many other similar projects in the area of high-confidence robotics and cyber-physical systems design, is to be able to reason formally about such systems.

A spectrum of computer-assisted formal methods techniques have been applied to reasoning about concurrent robotics, ranging from interactive theorem proving to automated analysis via model checking. These techniques provide a tradeoff between manual effort and the expressiveness of specifications and the strength of guarantees. Our choice using manually specified choreographies and motion primitive specifications and automatically checked type correctness attempts to explore a point in the design space that requires manual abstraction of motion and geometry but provides automated checks for the interaction. We build on a type-based foundation rather than global model checking to again reflect the tradeoff: our use of choreographies and projections from global to local types restricts the structure of programs that can be type checked but enables a more scalable *local* check for each process; in contrast, a model checking approach could lead to state-space explosion already at the level of concurrency.

There are a vast number of extensions and applications of session types and choreographies [Ancona et al. 2016; Gay and Ravera 2017; Hüttel et al. 2016], but little work bringing types to practical programming in the domain of robotics or cyber-physical systems. We discuss the most related work. Our starting point was the theory of motion session types and PGCD [Banusic et al. 2019; Majumdar et al. 2019], whose goals are similar to ours. We considerably extend the scope and expressiveness of motion session types: through *continuous-time* motion primitives and through separating conjunction for modular synchronisation.

Syntactic extensions of original global types [Honda et al. 2008] to represent more expressive communication structures have been studied, e.g., in [Lange et al. 2015], in the context of synthesis from communicating automata; in [Castagna et al. 2012] to include parallel and choices; in [Demangeon and Honda 2012] to represent nested global types. Our aim is to include the minimum syntax extension to [Honda et al. 2008] to represent separation and synchronisations best suited to implementing robotics applications. In turn, a combination of motion primitives and predicates required us to develop a novel and non-trivial data flow analysis of global choreographies for the well-formedness check.

Hybrid extensions to process algebras [Bergstra and Middelburg 2005; Liu et al. 2010; Lynch et al. 2003; Rounds and Song 2003] extend process algebras with hybrid behaviour and study classical concurrency issues such as process equivalences. Extensions to timed (but not hybrid) specifications in the π-calculus are studied in Bocchi et al. [2019, 2014] to express properties on times on top of *binary* session typed processes.

The theory of hybrid automata [Alur et al. 1995; Henzinger 1996] provides a foundation for verifying models of hybrid systems. The main emphasis in hybrid automaton research has been in defining semantics and designing model checking algorithms—too many to enumerate, see [Henzinger 1996; Platzer 2018]—and not on programmability. Assume guarantee reasoning for hybrid systems has been studied, e.g. [Benveniste et al. 2018; Nuzzo 2015; Nuzzo et al. 2015; Tripakis 2016]. These works do not consider programmability or choreography aspects.

Deductive verification for hybrid systems attempts to define logics and invariant-based reasoning to hybrid systems. Differential dynamic logic (dL) [Platzer 2018; Platzer and Tan 2018] is a general logical framework to deductively reason about hybrid systems. It extends *dynamic logic* with differential operators and shows sound and (relatively) complete axiomatisations for the logic. Keymaera [Fulton et al. 2015] and HHL Prover [Wang et al. 2015] are interactive theorem provers based on hybrid progam logics. These tools can verify complex properties of systems at the cost of intensive manual effort. In contrast, we explore a point in the design space with more automation but less expressiveness.

A well-studied workflow in high-confidence cyber-physical systems is model-based design (see, e.g., [Henzinger and Sifakis 2007] for an overview), where a system is constructed by successive refinement of an abstract model down to an implementation. An important problem in model-based design is to ensure property-preserving refinement: one verifies properties of the system at higher levels of abstraction, and ensures that properties are preserved through refinement. In the presence of continuous dynamics, defining an appropriate notion of refinement and proving property-preserving compilation formally are hard problems [Bohrer et al. 2018; Yan et al. 2020].

In our implementation, we use *automated* tools based on the dReal SMT solver [Gao et al. 2013]. Our verification is *semi-automatic*, as we require user annotations for footprints or for the motion primitive specifications, but discharge verification conditions through dReal. Of course, there are programs that go past the capability of the solver—this is already true for systems that *only* have concurrency or only deal with dynamics. Our proof rules is to allow sound reasoning, potentially inside an interactive prover. Our implementation shows that—at least some—nontrivial examples do allow automation. Non-linear arithmetic is difficult to scale. Our observation is that a combination of manual specification of *abstract* footprints that replace complex geometrical shapes with simpler over-approximations along with the power of state-of-the-art SMT solvers is reasonably effective even for complicated examples.

Our choice of dReal (as opposed to a different SMT solver) is dReal's "off the shelf" support for non-linear arithmetic and trigonometric functions. Trigonometry shows up in our handling of frame shifts. Note that dReal only considers δ-decidability and can be incomplete in theory. We expect that most implementations will be collision free in a "robust" way (that is, two robots will not just not collide, they would be separated by a minimum distance). Therefore we believe the potential incompleteness is less of a concern. Indeed, the main limiting factor in our experiments was scalability for larger verification conditions.

Formal methods have been applied to multi-robot planning [Desai et al. 2017; Gavran et al. 2017]. These systems ignore geometry and view robots as sequences of atomic motion primitives.

7 DISCUSSION AND FUTURE DIRECTIONS

In this paper, we have shown how choreographies can be extended with dynamic motion primitives to enable compositional reasoning in the presence of continuous-time dynamics. We have developed an automated verification tool and a compiler from type-correct programs to distributed robotics applications using ROS and commercial and custom-made robotics hardware. Our goal is to *integrate* types and static analysis techniques into existing robotics frameworks, rather than provide fully verified stacks (see, e.g., [Bohrer et al. 2018]). We explain our language features in terms of a calculus but session processes can be easily embedded into existing frameworks for robot programming. We have demonstrated that the language and type system are expressive enough to statically verify distributed manoeuvres on top of existing hardware and software.

We view our paper as a first step in verifying robotics programs. There are many other important but yet unmodelled aspects. We outline several directions not addressed in our work.

For example, we omit *probabilistic robotics* aspects, including the perception stack (vision, LIDAR, etc.), and aspects such as filtering, localisation, and mapping [LaValle 2012; Thrun et al. 2006]. These will require a probabilistic extension to our theory. Such a theory requires a nontrivial extension of program logics and analyses for probabilistic programs [McIver and Morgan 2005] with the verification and synthesis for stochastic continuous-state systems [Zamani et al. 2014]. We also omit any modeling of the perception stack or dynamic techniques, often based on machine learning, of learning the environment. Instead, our models *assume* worst-case disturbance bounds on the sensing or dynamics. We believe an integration of learning techniques with formal methods is an interesting challenge but goes beyond the scope of this paper.

Our framework statically verifies properties of a program. In practice, robots work in dynamic, often unknown, environments [LaValle 2006, 2012; Siegwart et al. 2011; Thrun et al. 2006]. When confronted with a formal method, domain experts often expect that a verification methodology should be able to verify correctness of behaviors in an *arbitrary* dynamic environment and any failure to do so simply shows the inadequacy of verification techniques. Formal methods cannot prove correctness in an *arbitrary* dynamic environment. When we verify a system, it is—as true in any formal methods—*relative* to an environment assumption; such assumptions are usually implicit in robotics implementations. Thus, we can model moving obstacles, etc. in the environment through assumptions on the behaviour of such obstacles (e.g., limits on their speed or trajectories); these assumptions are propagated by our assume-guarantee proof system, and show up as a premise in the eventual correctness proof.

We focus on communication safety and collision freedom as the basic correctness conditions any system has to satisfy. An interesting next step is to extend the reasoning to more expressive specifications. For safety specifications, such as invariants, one could reduce the problem to checking communication safety. For liveness specifications, the proof system would need to be extended with ranking arguments.

While all the above problems are interesting in their own right, they are orthogonal to our main contribution that one can reason about concurrency and dynamics in continuous time in a type-based setting. Our future work will look at more expressive scenarios, but the setting in our paper already required complex proofs and it was important for us to get the core correct. We believe a verification system that can faithfully model and uniformly reason about more complex interactions and that scales to larger implementations remains a grand challenge in computer science (see, e.g., [Lozano-Pérez 1983] for an articulation of these challenges).

ACKNOWLEDGMENTS

We thank the OOPSLA reviewers for useful comments and suggestions; and Julia Gabet for testing the artifact. Majumdar and Zufferey are supported in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248–CPEC and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT). Yoshida is supported in part by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

REFERENCES

M. Abadi and L. Lamport. 1993. Composing Specifications. *ACM Transactions on Programming Languages and Systems* 15, 1 (1993), 73–132.

R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. 1995. The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138 (1995), 3–34.

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. 2016. Behavioral Types in Programming Languages. *FTPL* 3(2-3) (2016), 95–230.

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

Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. 2018. Contracts for System Design. *Foundations and Trends in Electronic Design Automation* 12, 2-3 (2018), 124–400. https://doi.org/10.1561/1000000053

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

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

Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2019. Asynchronous Timed Session Types. In *28th European Symposium on Programming (LNCS)*, Vol. 11423. Springer, 583–610.

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

Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: verified controller executables from verified cyber-physical system models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 617–630. https://doi.org/10.1145/3192366.3192406

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2012. On Global Types and Multi-Party Session. *Logical Methods in Computer Science* 8, 1 (2012).

K.M. Chandy and J. Misra. 1988. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company.

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *23rd International Conference on Concurrency Theory (LNCS)*, Vol. 7454. Springer, 272–286.

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

Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2017. DRONA: a framework for safe distributed mobile robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems, ICCPS 2017, Pittsburgh, Pennsylvania, USA, April 18-20, 2017*, Sonia Martínez, Eduardo Tovar, Chris Gill, and Bruno Sinopoli (Eds.). 239–248. https://doi.org/10.1145/3055004.3055022

Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. 2015. Precise subtyping for synchronous multiparty sessions. In *PLACES (EPTCS)*, Vol. 203. 29–43. https://doi.org/10.4204/EPTCS.203.3

Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. 2016. 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 (LNCS)*, Vol. 9660. Springer, 155–172.

Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 527–538. https://doi.org/10.1007/978-3-319-21401-6_36

Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Automated Deduction - CADE-24 (Lecture Notes in Computer Science)*, Vol. 7898. Springer, 208–214.

Ivan Gavran, Rupak Majumdar, and Indranil Saha. 2017. Antlab: A Multi-Robot Task Server. *ACM Trans. Embedded Comput. Syst.* 16, 5s (2017), 190:1–190:19. https://doi.org/10.1145/3126513

Simon Gay and Antonio Ravera (Eds.). 2017. *Behavioural Types: from Theory to Tools.* River Publishers.

Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovi, Alceste Scalas, and Nobuko Yoshida. 2019a. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming* (2019). To appear.

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

2018.12.002

T.A. Henzinger. 1996. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 278–292.

Thomas A. Henzinger, Marius Minea, and Vinayak S. Prabhu. 2001. Assume-Guarantee Reasoning for Hierarchical Hybrid Systems. In *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings (Lecture Notes in Computer Science)*, Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli (Eds.), Vol. 2034. Springer, 275–290. https://doi.org/10.1007/3-540-45351-2_24

Thomas A. Henzinger and Joseph Sifakis. 2007. The Discipline of Embedded Systems Design. *IEEE Computer* 40, 10 (2007), 36–44.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. ACM Press, 273–284. https://doi.org/10.1145/1328438.1328472

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *JACM* 63 (2016), 1–67. Issue 1-9.

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. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (2016). https://doi.org/10.1145/2873052

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619. https://doi.org/10.1145/69575.69577

Dimitrios Kouzapas and Nobuko Yoshida. 2013. Globally Governed Session Semantics. In *CONCUR (LNCS)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.), Vol. 8052. Springer, 395–409. https://doi.org/10.1145/1328438.1328472

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

Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. In *POPL*. ACM, 221–232.

Steven M. LaValle. 2006. *Planning Algorithms*. Cambride University Press.

Steven M. LaValle. 2012. *Sensing and Filtering*. NOW Publishers.

Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. 2010. A Calculus for Hybrid CSP. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science)*, Vol. 6461. Springer, 1–15.

Tomás Lozano-Pérez. 1983. Robot programming. *Proc. IEEE* 71, 7 (1983), 821–841.

Nancy A. Lynch, Roberto Segala, and Frits W. Vaandrager. 2003. Hybrid I/O automata. *Inf. Comput.* 185, 1 (2003), 105–157. https://doi.org/10.1016/S0890-5401(03)00067-1

Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. 2019. Motion Session Types for Robotic Interactions. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19) (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Rupak Majumdar, Nobuko Yoshida, and Damien Zufferey. 2020. Multiparty Motion Coordination: From Choreographies to Robotics Programs (Full version). arXiv:cs.RO/2010.05484

Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.

Pierluigi Nuzzo. 2015. *Compositional Design of Cyber-Physical Systems Using Contracts*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html

Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, and Tiziano Villa. 2015. A Platform-Based Design Methodology With Contracts and Related Tools for the Design of Cyber-Physical Systems. *Proc. IEEE* 103, 11 (2015), 2104–2132. https://doi.org/10.1109/JPROC.2015.2453253

André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer. https://doi.org/10.1007/978-3-319-63588-0

André Platzer and Yong Kiam Tan. 2018. Differential Equation Axiomatization: The Impressive Power of Differential Ghosts. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. ACM, 819–828.

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

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. ACM, 159–169.

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

Alceste Scalas and Nobuko Yoshida. 2019. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.* 3, POPL, Article 30 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290343

Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. 2011. *Introduction to Autonomous Mobile Robots*. MIT.

Sebastian Thrun, Wolfram Burgard, and Dieter Fox. 2006. *Probabilistic Robotics*. MIT.

Stavros Tripakis. 2016. Compositionality in the Science of System Design. *Proc. IEEE* 104, 5 (2016), 960–972. https://doi.org/10.1109/JPROC.2015.2510366

Shuling Wang, Naijun Zhan, and Liang Zou. 2015. An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings (Lecture Notes in Computer Science)*, Vol. 9407. Springer, 382–399.

Gaogao Yan, Li Jiao, Shuling Wang, Lingtai Wang, and Naijun Zhan. 2020. Automatically Generating SystemC Code from HCSP Formal Models. *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 4:1–4:39. https://doi.org/10.1145/3360002

Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *16th International Conference on Distributed Computing and Internet Technology (LNCS)*, Vol. 11969. Springer, 73–93.

Majid Zamani, Peyman Mohajerin Esfahani, Rupak Majumdar, Alessandro Abate, and John Lygeros. 2014. Symbolic Control of Stochastic Systems via Approximately Bisimilar Finite Abstractions. *IEEE Trans. Automat. Contr.* 59, 12 (2014), 3135–3150. https://doi.org/10.1109/TAC.2014.2351652