

# 1 Motion Session Types for Robotic Interactions

2 **Rupak Majumdar**

3 MPI-SWS, Germany

4 rupak@mpi-sws.org

5 **Marcus Pirron**


6 MPI-SWS, Germany

7 mpirron@mpi-sws.org

8 **Nobuko Yoshida** 

9 Imperial College London, UK

10 n.yoshida@imperial.ac.uk

11 **Damien Zufferey** 

12 MPI-SWS, Germany

13 zufferey@mpi-sws.org

---

## 14 — Abstract —

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

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

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

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

36 **2012 ACM Subject Classification** Computer systems organization → Robotics; Software and its  
37 engineering → Concurrent programming languages; Theory of computation → Process calculi;  
38 Theory of computation → Type theory

39 **Keywords and phrases** Session Types, Robotics, Concurrent Programming, Motions, Communica-  
40 tions, Multiparty Session Types, Deadlock Freedom

41 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2019.12

42 **Category** Brave New Idea Paper

43 **Funding** *Rupak Majumdar*: DFG 389792660 TRR 248, ERC Synergy Grant 610150

44 *Marcus Pirron*: DFG 389792660 TRR 248, ERC Synergy Grant 610150

45 *Nobuko Yoshida*: EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and  
46 EP/N028201/1

47 *Damien Zufferey*: DFG 389792660 TRR 248, ERC Synergy Grant 610150



© Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey;  
licensed under Creative Commons License CC-BY

33rd European Conference on Object-Oriented Programming (ECOOP 2019).

Editor: Alastair F. Donaldson; Article No. 12; pp. 12:1–12:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

48 **1** Introduction

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

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

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

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

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

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

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

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

## 108 **2 A Gentle Introduction to Motion Session Types**

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

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

120 Motion session types extend session types, introduced in a series of papers during the  
121 1990s [23, 43, 24], in the context of pure concurrent programming. Session types have since  
122 been studied in many contexts over the last decade—see the surveys of the field [27, 17].

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

### 126 **2.1 Communication: Multiparty Session Types**

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

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

134 Specifically, the protocol works as follows.

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

## 12:4 Motion Session Types for Robotic Interactions

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

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

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

$$152 \quad \mu t. \text{Cart} \rightarrow \text{Arm} : \{ \text{fold. Arm} \rightarrow \text{Cart} : \text{ok. Cart} \rightarrow \text{Arm} : \text{grab. Arm} \rightarrow \text{Cart} : \text{ok.t}, \text{done.end} \} \quad (1)$$

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

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

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

$$165 \quad \mu t. ( ( !\text{fold}. ?\text{ok}. !\text{grab}. ?\text{ok.t} ) \oplus ( !\text{done}. \text{end} ) ) \quad (2)$$

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

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

$$171 \quad \mu t. ( ( ?\text{fold}. !\text{ok}. ?\text{grab}. !\text{ok.t} ) \& ( ?\text{done}. \text{end} ) ) \quad (3)$$

172 in which  $\&$  means that a choice is offered externally.

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

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

179 For example, consider a simple ring protocol, where the *Arm* process above is divided into  
 180 two parts, *Lower* and *Upper*. Now, *Cart* sends a message *fold* to the lower arm *Lower*, which

181 forwards the message to *Upper*. After receiving the message, *Upper* sends an acknowledgement  
 182 *ok* to *Cart*. We start by specifying the global type as:

$$183 \quad \text{Cart} \rightarrow \text{Lower} : \text{fold} . \text{Lower} \rightarrow \text{Upper} : \text{fold} . \text{Upper} \rightarrow \text{Cart} : \text{ok} . \text{end} \quad (4)$$

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

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

$$\begin{aligned} & \text{Cart's endpoint type:} && \text{Lower!fold.Upper?ok.end} \\ 187 & \text{Lower's endpoint type:} && \text{Cart?fold.Upper!fold.end} \\ & \text{Upper's endpoint type:} && \text{Lower?fold.Cart!ok.end} \end{aligned}$$

188 where *Lower!fold* means “send to *Lower* a *fold* message,” and *Upper?ok* means “receive from  
 189 *Upper* an *ok* message.” Then each process is type-checked against its own endpoint type.  
 190 When the three processes are executed, their interactions automatically follow the stipulated  
 191 scenario.

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

## 201 2.2 Motion: Motion Primitives and Trajectories

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

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

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

$$\begin{aligned} 217 & \text{Cart} \rightarrow \text{Arm} : \text{fold} . \text{dt}\langle \text{Cart} : \text{idle}, \text{Arm} : \text{fold} \rangle . \\ 218 & \text{Arm} \rightarrow \text{Cart} : \text{ok} . \text{dt}\langle \text{Cart} : \text{move}, \text{Arm} : \text{idle} \rangle . G \end{aligned}$$

220 where “ $\text{dt}\langle \text{Cart} : \text{idle}, \text{Arm} : \text{fold} \rangle$ ” specifies the joint motion primitives *idle* executed by the  
 221 *Cart* and *fold* executed by the *Arm* are synchronised. We extend local types with motion

## 12:6 Motion Session Types for Robotic Interactions

222 primitives as well. The conformance check ensures that, if each process conforms to its  
223 local types, then the composition of the system conforms to the global type—which now  
224 includes both message-based synchronization as well as synchronization over time using  
225 motion primitives.

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

### 231 **3 Motion Session Calculus**

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

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

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

#### 247 **Motion Primitives**

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

#### 258 **Syntax of Motion Session Calculus**

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



262 calculus are defined by:

$$263 \quad P ::= \mathfrak{p}!\ell\langle e \rangle.P \mid \sum_{i \in I} \mathfrak{p}?\ell_i(x_i).P_i \mid \sum_{i \in I} \mathfrak{p}?\ell_i(x_i).P_i + \mathfrak{dt}\langle a \rangle.P \mid \mathfrak{dt}\langle a \rangle.P \\ \mid \text{if } e \text{ then } P \text{ else } Q \mid \mu X.P \mid X \mid \mathbf{0}$$

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

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

$$277 \quad M ::= \mathfrak{p} \triangleleft P \mid M \mid M$$

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

## 283 Operational Semantics of Motion Session Calculus

284 The value  $v$  of expression  $e$  (notation  $e \downarrow v$ ) is computed as expected. We assume that  $e \downarrow v$   
 285 is effectively computable and takes logical “zero time.”

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

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

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

303 The motion primitives are handled with [MOTION] and [M-PAR]. Here, we need to label  
 304 transitions with the time taken by the action and propagate these labels with the parallel

## 12:8 Motion Session Types for Robotic Interactions

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

$$\begin{array}{c}
 \text{[COMM]} \\
 \frac{j \in I \quad e \downarrow v}{\mathfrak{p} \triangleleft \sum_{i \in I} \mathfrak{q} ? \ell_i(x). P_i + \text{dt}\langle a \rangle . P \mid \mathfrak{q} \triangleleft \mathfrak{p} ! \ell_j \langle e \rangle . Q \longrightarrow \mathfrak{p} \triangleleft P_j \{v/x\} \mid \mathfrak{q} \triangleleft Q} \\
 \\
 \begin{array}{cc}
 \text{[DEFAULT]} & \text{[MOTION]} \\
 \mathfrak{p} \triangleleft \sum_{i \in I} \mathfrak{q} ? \ell_i(x). P_i + \text{dt}\langle a \rangle . P \xrightarrow{\text{dt}\langle a \rangle} \mathfrak{p} \triangleleft P & \mathfrak{p} \triangleleft \text{dt}\langle a \rangle . P \xrightarrow{\text{dt}\langle a \rangle} \mathfrak{p} \triangleleft P
 \end{array} \\
 \\
 \begin{array}{cc}
 \text{[T-CONDITIONAL]} & \text{[F-CONDITIONAL]} \\
 \frac{e \downarrow \text{true}}{\mathfrak{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \longrightarrow \mathfrak{p} \triangleleft P} & \frac{e \downarrow \text{false}}{\mathfrak{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \longrightarrow \mathfrak{p} \triangleleft Q}
 \end{array} \\
 \\
 \begin{array}{cc}
 \text{[R-PAR]} & \text{[M-PAR]} \\
 \frac{\mathfrak{p} \triangleleft Q \longrightarrow \mathfrak{p} \triangleleft Q'}{\mathfrak{p} \triangleleft Q \mid M \longrightarrow \mathfrak{p} \triangleleft Q' \mid M} & \frac{\mathfrak{p}_i \triangleleft P_i \xrightarrow{\text{dt}\langle a_i \rangle} \mathfrak{p}_i \triangleleft P'_i \quad \forall i, j. \text{duration}(a_i) = \text{duration}(a_j)}{\prod_i \mathfrak{p}_i \triangleleft P_i \xrightarrow{\text{dt}\langle (\mathfrak{p}_i : a_i) \rangle} \prod_i \mathfrak{p}_i \triangleleft P'_i}
 \end{array} \\
 \\
 \begin{array}{cc}
 \text{[R-STRUCT]} & \text{[M-STRUCT]} \\
 \frac{M'_1 \equiv M_1 \quad M_1 \longrightarrow M_2 \quad M_2 \equiv M'_2}{M'_1 \longrightarrow M'_2} & \frac{M'_1 \equiv M_1 \quad M_1 \xrightarrow{\text{dt}\langle a \rangle} M_2 \quad M_2 \equiv M'_2}{M'_1 \xrightarrow{\text{dt}\langle a \rangle} M'_2}
 \end{array}
 \end{array}$$



<pre> Cart&lt;   Arm!fold().   wait (dt&lt;idle&gt;){     Arm?ok().     dt&lt;move&gt;.     Arm!grab.     wait (dt&lt;idle&gt;){       Arm?ok().       dt&lt;move&gt;.       Arm!done().0     }   } </pre>	<pre> Arm&lt;   μX.wait (dt&lt;idle&gt;){     Cart?fold().dt&lt;fold&gt;.Cart!ok().X   + Cart?grab().     dt&lt;grip&gt;.     Cart!ok().X   + Cart?done().0   } </pre>
--	--

■ **Figure 1** A cart and arm example

305 composition. This ensures that when (physical) time elapses for one process, it elapses  
 306 equally for all processes; every process has to spend the same amount of time. This style of  
 307 synchronisation is reminiscent of broadcast calculi [39]. Instead of broadcast messages, we  
 308 broadcast *time*.

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

311 ► **Definition 3.2.** *A multiparty motion session  $M$  is stuck if  $M \neq \prod_{i \in I} \mathbf{p}_i \triangleleft \mathbf{0}$  and there is*  
 312 *no multiparty session  $M'$  such that  $M \longrightarrow M'$ . A multiparty session  $M$  gets stuck, notation*  
 313  *$\mathbf{stuck}(M)$ , if it reduces to a stuck motion multiparty session.*

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

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

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

330 The motion primitive *idle* keeps the cart or the arm stationary. The primitive *move* moves  
 331 the cart, the primitives *grip* and *fold* respectively move the arm to grab an object or to fold

PGCD: pseudo code for the Arm	Arm $\triangleleft$
1 <b>while</b> <i>true</i> <b>do</b>	$\mu X.$ wait (dt(idle)){
2   <b>receive</b> (idle)	Cart? <i>fold</i> ().
3       <i>fold</i> $\Rightarrow$	dt(fold).
4           <b>fold</b> ();	Cart! <i>ok</i> () $\cdot X$
5               <b>send</b> (Cart, <i>ok</i> )	+ Cart? <i>grab</i> ().
6               <i>grab</i> $\Rightarrow$	dt(grip).
7                   <b>grip</b> ();	Cart! <i>ok</i> () $\cdot X$
8                       <b>send</b> (Cart, <i>ok</i> )	+ Cart? <i>done</i> ().
9                           <i>done</i> $\Rightarrow$	<b>0</b>
10                               <b>break</b>	}

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

332 the arm. At this point, we focus on the communication pattern and therefore abstract away  
 333 the actual trajectories traced by the motion primitives. We come back to the trajectories in  
 334 Section 5.

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

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

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

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

## 352 4 Multiparty Motion Session Types

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

### 355 4.1 Motion Session Types and Projections

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

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

■ **Figure 3** A multi-party handover example.

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

359  $S ::= \text{unit} \mid \text{nat} \mid \text{int} \mid \text{bool} \mid \text{real}$

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

361  $G ::= \text{dt}\langle(p_i : a_i)\rangle.G \mid p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I} \mid \mathbf{t} \mid \mu\mathbf{t}.G \mid \text{end}$

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

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

374 ► **Example 4.2** (Global session types). The global session type for the fetch example (Ex-

## 12:12 Motion Session Types for Robotic Interactions

ample 3.3) is:

```

376     Cart → Arm : fold(unit).dt⟨Cart : idle, Arm : fold⟩.
377     Arm → Cart : ok(unit).dt⟨Cart : move, Arm : idle⟩.
378     Cart → Arm : grab(unit).dt⟨Cart : idle, Arm : grip⟩.
379     Arm → Cart : ok(unit).dt⟨Cart : move, Arm : idle⟩.
380     Cart → Arm : done(unit).end
381

```

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

```

383     Cart → Arm : fold(unit).dt⟨Cart : idle, Carrier : idle, Arm : fold⟩.
384     Arm → Cart : ok(unit).Cart → Carrier : ok(unit).
385     dt⟨Cart : move, Carrier : move, Arm : idle⟩.
386     Carrier → Cart : ok(unit).Cart → Arm : grab(unit).
387     dt⟨Cart : idle, Carrier : idle, Arm : grip⟩.
388     Arm → Cart : ok(unit).Cart → Carrier : ok(unit).
389     dt⟨Cart : move, Carrier : move, Arm : idle⟩.
390     Cart → Arm : done(unit).Cart → Carrier : done(unit).end
391

```

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

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

$$\begin{aligned}
 T & ::= dt\langle a \rangle.T \mid \&\{p?\ell_i(S_i).T_i\}_{i \in I} \mid \&\{p?\ell_i(S_i).T_i\}_{i \in I} \& dt\langle a \rangle.T \mid \oplus\{q!\ell_i(S_i).T_i\}_{i \in I} \\
 & \mid \mathbf{t} \mid \mu\mathbf{t}.T \mid \mathbf{end}
 \end{aligned}$$

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

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

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

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

- $G = \text{end}$  implies  $T = \text{end}$ ; [PROJ-END]
- $G = \text{dt}\langle p_i : a_i \rangle . G'$  implies  $T = \text{dt}\langle a_j \rangle . T'$  with  $r = p_j$  and  $G' \downarrow_r T'$ ; [PROJ-MOTION]
- $G = p \rightarrow r : \{\ell_i(S_i).G_i\}_{i \in I}$  implies  $T = \&\{p?\ell_i(S_i).T_i\}_{i \in I}$  with  $G_i \downarrow_r T_i$ ; [PROJ-IN]
- 419 •  $G = r \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$  implies  $T = \oplus\{q!\ell_i(S_i).T_i\}_{i \in I}$  and  $G_i \downarrow_r T_i, \forall i \in I$ ; [PROJ-OUT]
- $G = p \rightarrow q : \{\ell_i(S_i).G_i\}_{i \in I}$  and  $r \notin \{p, q\}$  implies that there are  $T_i, i \in I$  s.t. [PROJ-CONT]  
 $T = \prod_{i \in I} T_i$ , and  $G_i \downarrow_r T_i$ , for every  $i \in I$ .
- $G = \mu t.G$  implies  $T = \mu t.T'$  with  $G \downarrow_r T'$  if  $r$  occurs in  $G$ , otherwise  $T = \text{end}$ . [PROJ-REC]

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

$$421 \quad T_1 \prod T_2 = \begin{cases} T_1 & \text{if } T_1 = T_2 & \text{[MRG-ID]} \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} & \text{and} \\ T_2 = \&\{p'?\ell_j(S_j).T_j\}_{j \in J} & \text{and} \\ T_3 = \&\{p'?\ell_k(S_k).T_k\}_{k \in I \cup J} \end{cases} & \text{[MRG-BRA1]} \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} \& \text{dt}\langle a \rangle . T' & \text{and} \\ T_2 = \&\{p'?\ell_j(S_j).T_j\}_{j \in J} \& \text{dt}\langle a \rangle . T' & \text{and} \\ T_3 = \&\{p'?\ell_k(S_k).T_k\}_{k \in I \cup J} \& \text{dt}\langle a \rangle . T' \end{cases} & \text{[MRG-BRA2]} \\ 421 \quad T_3 & \text{if } \exists I, J : \begin{cases} T_1 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} & \text{and} \\ T_2 = \&\{p'?\ell_j(S_j).T_j\}_{j \in J} \& \text{dt}\langle a \rangle . T' & \text{and} \\ T_3 = \&\{p'?\ell_k(S_k).T_k\}_{k \in I \cup J} \& \text{dt}\langle a \rangle . T' \end{cases} & \text{[MRG-BRA3]} \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = \text{dt}\langle a \rangle . T' & \text{and} \\ T_2 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} \& \text{dt}\langle a \rangle . T' & \text{and} \\ T_3 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} \& \text{dt}\langle a \rangle . T' \end{cases} & \text{[MRG-BRA4]} \\ T_3 & \text{if } \exists I, J : \begin{cases} T_1 = \text{dt}\langle a \rangle . T' & \text{and} \\ T_2 = \&\{p'?\ell_j(S_j).T_j\}_{j \in I} & \text{and} \\ T_3 = \&\{p'?\ell_i(S_i).T_i\}_{i \in I} \& \text{dt}\langle a \rangle . T' \end{cases} & \text{[MRG-BRA5]} \\ T_2 \prod T_1 & \text{if } T_2 \prod T_1 \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

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

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

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

429  $\text{Arm!fold}\langle \text{unit} \rangle . \text{dt}\langle \text{idle} \rangle . \text{Arm?ok}\langle \text{unit} \rangle . \text{dt}\langle \text{move} \rangle . \text{Arm!grab}\langle \text{unit} \rangle .$   
 430  $\text{dt}\langle \text{idle} \rangle . \text{Arm?ok}\langle \text{unit} \rangle . \text{dt}\langle \text{move} \rangle . \text{Arm!done}\langle \text{unit} \rangle . \text{end}$   
 431

432 The local motion session type for the arm is:

433  $\text{Cart?fold}\langle \text{unit} \rangle . \text{dt}\langle \text{fold} \rangle . \text{Cart!ok}\langle \text{unit} \rangle . \text{dt}\langle \text{idle} \rangle . \text{Cart?grab}\langle \text{unit} \rangle .$   
 434  $\text{dt}\langle \text{grip} \rangle . \text{Cart!ok}\langle \text{unit} \rangle . \text{dt}\langle \text{idle} \rangle . \text{Cart?done}\langle \text{unit} \rangle . \text{end}$   
 435

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

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

448 This problem is not unique to our system but a more general problem in defining the  
 449 semantics of hybrid systems [20, 21]. In general, one needs to assume that time always  
 450 *diverges* for infinite executions. In this work, we take a pragmatic solution and simply *disallow*  
 451 *0-time recursion*. When recursion is used, all the paths between a  $\mu\mathbf{t}$  and the corresponding  $\mathbf{t}$   
 452 must contain at least one motion primitive. This is a simple check which can be done at the  
 453 syntactic level of global types and it is a sufficient condition for forcing the progress of time.

## 454 4.2 Motion Session Typing

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

$$457 \quad \Gamma \vdash e : S \quad \Gamma \vdash P : T \quad \vdash M : G$$

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

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

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

$$\begin{array}{c}
 \text{[SUB-END]} \quad \text{[SUB-IN1]} \\
 \text{end} \leq \text{end} \quad \frac{\forall i \in I : S'_i \leq S_i \quad T_i \leq T'_i \quad T \leq T'}{\&\{p?l_i(S_i).T_i\}_{i \in I \cup J} \& \text{dt}\langle a \rangle.T \leq \&\{p?l_i(S'_i).T'_i\}_{i \in I} \& \text{dt}\langle a \rangle.T'} \\
 \\
 \text{[SUB-MOTION]} \quad \text{[SUB-IN2]} \\
 \frac{T \leq T'}{\text{dt}\langle a \rangle.T \leq \text{dt}\langle a \rangle.T'} \quad \frac{\forall i \in I : S'_i \leq S_i \quad T_i \leq T'_i}{\&\{p?l_i(S_i).T_i\}_{i \in I \cup J} \& \text{dt}\langle a \rangle.T \leq \&\{p?l_i(S'_i).T'_i\}_{i \in I}} \\
 \\
 \text{[SUB-IN3]} \quad \text{[SUB-OUT]} \\
 \frac{T \leq T'}{\&\{p?l_i(S_i).T_i\}_{i \in I} \& \text{dt}\langle a \rangle.T \leq \text{dt}\langle a \rangle.T'} \quad \frac{\forall i \in I : S_i \leq S'_i \quad T_i \leq T'_i}{\oplus\{p!l_i(S_i).T_i\}_{i \in I} \leq \oplus\{p!l_i(S'_i).T'_i\}_{i \in I \cup J}}
 \end{array}$$

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

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

- 471 ■ [T-SUB] is the *subsumption rule*: a process with type  $T$  is also typed by the supertype  $T'$ ;
- 472 ■ [T-0] says that a terminated process implements the terminated session type;
- 473 ■ [T-REC] types a recursive process  $\mu X.P$  with  $T$  if  $P$  can be typed as  $T$ , too, by extending  
474 the typing environment with the assumption that  $X$  has type  $T$ ;
- 475 ■ [T-VAR] uses the typing environment assumption that process  $X$  has type  $T$ ;
- 476 ■ [T-MOTION] types a motion process as a motion local type;
- 477 ■ [T-INPUT-CHOICE] types a summation of input prefixes as a branching type and a default  
478 branch as a motion type. It requires that each input prefix targets the same participant  
479  $q$ , and that, for all  $i \in I$ , each continuation process  $P_i$  is typed by the continuation type  
480  $T_i$ , having the bound variable  $x_i$  in the typing environment with sort  $S_i$ . Note that the  
481 rule implicitly requires the process labels  $\ell_i$  to be pairwise distinct (as per Definition 4.3);
- 482 ■ [T-OUT] types an output prefix with a singleton selection type, provided that the expression  
483 in the message payload has the correct sort  $S$ , and the process continuation matches the  
484 type continuation;
- 485 ■ [T-CHOICE] types a conditional process by matching the branches of the types to branches  
486 of the sub-processes;
- 487 ■ [T-SESS] types multiparty sessions, by associating typed processes to participants. It  
488 requires that the processes being composed in parallel can play as participants of a  
489 global communication protocol: hence, their types must be projections of a single global  
490 type  $G$ . As the temporal evolution (motion) synchronises all the processes condition  
491  $\text{pt}\{G\} = \{p_i \mid i \in I\}$  guarantees that motions are defined for every participant.

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

500  $\text{dt}\langle \text{idle} \rangle. \text{Cart?grab}(\text{unit}).T$

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

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

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

509 ► **Definition 4.8** (Global types consumption and reduction). *The consumption of the commu-*  
510 *nication*  $p \xrightarrow{\ell} q$  *and motion*  $\text{dt}\langle a \rangle$  *for the global type*  $G$  *(notation*  $G \setminus p \xrightarrow{\ell} q$  *and*  $G \setminus \text{dt}\langle a \rangle$  *) is*



## 12:16 Motion Session Types for Robotic Interactions

■ **Table 2** Typing rules for motion processes.

$\frac{[\text{T-0}]}{\Gamma \vdash \mathbf{0} : \text{end}}$	$\frac{[\text{T-REC}]}{\Gamma, X : T \vdash P : T} \quad \Gamma \vdash \mu X.P : T$	$\frac{[\text{T-VAR}]}{\Gamma, X : T \vdash X : T}$	$\frac{[\text{T-MOTION}]}{\Gamma \vdash Q : T} \quad \Gamma \vdash \text{dt}\langle a \rangle.Q : \text{dt}\langle a \rangle.T$
$\frac{[\text{T-OUT}]}{\Gamma \vdash \mathbf{e} : S \quad \Gamma \vdash P : T} \quad \Gamma \vdash \mathbf{q}!\ell(\mathbf{e}).P : \mathbf{q}!\ell(S).T$	$\frac{[\text{T-INPUT-CHOICE1}]}{\Gamma \vdash \sum_{i \in I} \mathbf{q}?\ell_i(x_i).P_i : \&\{\mathbf{q}?\ell_i(S_i).T_i\}_{i \in I}}$		
$\frac{[\text{T-INPUT-CHOICE2}]}{\Gamma \vdash \sum_{i \in I} \mathbf{q}?\ell_i(x_i).P_i + \text{dt}\langle a \rangle.Q : \&\{\mathbf{q}?\ell_i(S_i).T_i\}_{i \in I} \& T} \quad \forall i \in I \quad \Gamma, x_i : S_i \vdash P_i : T_i \quad \Gamma \vdash \text{dt}\langle a \rangle.Q : T$			
$\frac{[\text{T-CHOICE}]}{\Gamma \vdash \text{if } \mathbf{e} \text{ then } P_1 \text{ else } P_2 : \oplus\{T_i\}_{i \in I \setminus \{k\}}} \quad \Gamma \vdash \mathbf{e} : \text{bool} \quad \exists k \in I \quad \Gamma \vdash P_1 : T_k \quad \Gamma \vdash P_2 : \oplus\{T_i\}_{i \in I \setminus \{k\}}$			
$\frac{[\text{T-SUB}]}{\Gamma \vdash P : T \quad T \leq T'} \quad \Gamma \vdash P : T'$	$\frac{[\text{T-SESS}]}{\vdash \prod_{i \in I} \mathbf{p}_i \triangleleft P_i : G} \quad \forall i \in I \quad \vdash P_i : G   \mathbf{p}_i \quad \text{pt}\{G\} = \{\mathbf{p}_i \mid i \in I\}$		

511 the global type defined (up to unfolding of recursive types) as follows:

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

$$512 \quad (\mathbf{p} \rightarrow \mathbf{q} : \{\ell_i(S_i).G_i\}_{i \in I}) \setminus \mathbf{p} \xrightarrow{\ell} \mathbf{q} = G_k \quad \text{if } \exists k \in I : \ell = \ell_k$$

$$513 \quad (\mathbf{r} \rightarrow \mathbf{s} : \{\ell_i(S_i).G_i\}_{i \in I}) \setminus \mathbf{p} \xrightarrow{\ell} \mathbf{q} = \mathbf{r} \rightarrow \mathbf{s} : \{\ell_i(S_i).G_i \setminus \mathbf{p} \xrightarrow{\ell} \mathbf{q}\}_{i \in I}$$

$$514 \quad \text{if } \{\mathbf{r}, \mathbf{s}\} \cap \{\mathbf{p}, \mathbf{q}\} = \emptyset \wedge \forall i \in I : \{\mathbf{p}, \mathbf{q}\} \subseteq G_i$$

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

515 We can now state the main results.

516 ► **Theorem 4.9** (Subject Reduction). *Let  $\vdash M : G$ .*

517 1. *For all  $M'$ , if  $M \longrightarrow M'$ , then  $\vdash M' : G'$  for some  $G'$  such that  $G \Longrightarrow G'$  or  $G = G'$ .*

518 2. *For all  $M'$ , if  $M \xrightarrow{\text{dt}\langle a \rangle} M'$ , then  $\vdash M' : G'$  for some  $G'$  such that  $G \Longrightarrow G'$ .*

519 ► **Corollary 4.10.** *Let  $\vdash M : G$ . If  $M \longrightarrow^* M'$ , then  $\vdash M' : G'$  for some  $G'$  such that*  
520  *$G \Longrightarrow G'$  or  $G = G'$ . Similarly for the case of  $\xrightarrow{\text{dt}\langle a \rangle}$ .*

521 ► **Theorem 4.11** (Progress). *If  $\vdash M : G$ , then either  $M \equiv \prod_{i \in I} \mathbf{p}_i \triangleleft \mathbf{0}$  or there is  $M'$  such*  
522 *that  $M \longrightarrow M'$  or  $M \xrightarrow{\text{dt}\langle a \rangle} M'$ .*

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

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

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

## 527 **5 Motion Primitives: Trajectories and Resources**

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

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

### 540 **5.1 Model for the Robots and Motion Primitives**

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

542 **Robots.** Each participant  $(p, q, \dots)$  maintains a state in the physical world. This state is  
543 updated when its own motion primitives execute as well as on potential physical interactions  
544 with other processes.

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

551 When two robots  $p_1$  and  $p_2$  are in the same environment, we may connect some state  
552 variables of one process to the external inputs of the other. This represents physical coupling  
553 between these robots. A *connection*  $\theta$  between  $p_1$  and  $p_2$  is a finite set of pairs of variables,  
554  $\theta = \{(x_i, w_i) \mid i = 1, \dots, m\}$ , such that: (1) for each  $(x, w) \in \theta$ , we have  $x \in p_1.X$  and  
555  $w \in p_2.W$  or  $x \in p_2.X$  and  $w \in p_1.W$ , and (2) there does not exist  $(x, w), (x', w) \in \theta$  such  
556 that  $x$  and  $x'$  are distinct. Two connections  $\theta_1$  and  $\theta_2$  are *compatible* if  $\theta_1 \cup \theta_2$  is a connection.  
557 We assume that all the participants in a session are connected by compatible connections.

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

## 12:18 Motion Session Types for Robotic Interactions

566 **Motion Primitives.** Let  $X$  and  $W$  be two sets of real-valued variables, representing internal  
 567 state and external input variables of a robotic system, respectively. A motion primitive  
 568 updates the values of the variables in  $X$  over time, while respecting the values of variables  
 569 in  $W$  set by the external world. This dynamic process results in a pair of state and input  
 570 trajectories  $(\xi, \nu)$ , i.e., a valuation over time to variables in  $X$  and  $W$ .

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

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

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

$$\begin{array}{c}
 \text{[T-CONDITIONAL]} \qquad \qquad \qquad \text{[F-CONDITIONAL]} \\
 \frac{\rho(\mathbf{e}) \downarrow \text{true}}{\mathfrak{p}, \rho \triangleleft \text{if } \mathbf{e} \text{ then } P \text{ else } Q \longrightarrow \mathfrak{p}, \rho \triangleleft P} \qquad \frac{\rho(\mathbf{e}) \downarrow \text{false}}{\mathfrak{p}, \rho \triangleleft \text{if } \mathbf{e} \text{ then } P \text{ else } Q \longrightarrow \mathfrak{p}, \rho \triangleleft Q} \\
 581
 \end{array}$$

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

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

$$\begin{array}{c}
 \text{[MOTION]} \\
 \frac{a = (T, \text{Pre}, \text{Inv}, \text{Post}) \quad \text{range}(\xi) = [0, T] \quad \rho = \xi(0) \quad \rho' = \xi(T) \\
 (\xi(0), \nu(0)) \in \text{Pre} \quad (\xi(T), \nu(T)) \in \text{Post} \quad \forall t \in [0, T]. (\xi(t), \nu(t)) \in \text{Inv}}{\mathfrak{p}, \rho \triangleleft \text{dt}(a).P \xrightarrow{\text{dt}(\langle \xi, \nu \rangle)} \mathfrak{p}, \rho' \triangleleft P} \\
 585
 \end{array}$$

586 The rule checks that the trajectory is valid w.r.t.  $a$ : the duration of the trajectory must  
 587 match the duration of the motion primitive, the start and end of the trajectory match  
 588 the state of  $\rho$  and  $\rho'$  respectively. Furthermore, the pre-condition, post-condition, and  
 589 invariant must be respected.

- 590 ■ The parallel composition of motions connects the external inputs of each process according  
 591 to the connections. For the notations, we use subscript to denote that an element belongs  
 592 to a particular process  $\mathfrak{p}$ , e.g.,  $X_{\mathfrak{p}}$  for the internal variables of  $\mathfrak{p}$ . We denote the restriction  
 593 of a trajectory  $\xi$  over a subset  $X$  of the dimensions by  $\xi|_X$ .

$$\begin{array}{c}
 \text{[M-PAR]} \\
 \frac{\forall i \quad \xi_i = \xi|_{X_{\mathfrak{p}_i}} \quad \nu_i = \theta_{\mathfrak{p}_i}(\xi)|_{W_{\mathfrak{p}_i}} \quad \mathfrak{p}_i, \rho_i \triangleleft P_i \xrightarrow{\text{dt}(\langle \xi_i, \nu_i \rangle)} \mathfrak{p}_i, \rho'_i \triangleleft P'_i \\
 \forall i, j, t. i \neq j \Rightarrow \text{rsrc}_{\mathfrak{p}_i}(\xi|_{X_{\mathfrak{p}_i}}(t), \theta_{\mathfrak{p}_i}(\xi)|_{W_{\mathfrak{p}_i}}(t)) \cap \text{rsrc}_{\mathfrak{p}_j}(\xi|_{X_{\mathfrak{p}_j}}(t), \theta_{\mathfrak{p}_j}(\xi)|_{W_{\mathfrak{p}_j}}(t)) = \emptyset}{\prod_i \mathfrak{p}_i, \rho_i \triangleleft P_i \xrightarrow{\text{dt}(\langle \xi, \nu \rangle)} \prod_i \mathfrak{p}_i, \rho'_i \triangleleft P'_i} \\
 594
 \end{array}$$

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

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

604 A trivial motion primitive **idle**( $\mathbf{p}_0, \mathbf{r}_0$ ) keeps the cart at its current position  $\mathbf{p}_0$  and  
 605 orientation  $\mathbf{r}_0$ ; the pre-condition is  $\mathbf{s}_{\text{Cart}} = 0$  (i.e., it is at rest), the post-condition is  
 606  $\mathbf{s}_{\text{Cart}} = 0 \wedge \mathbf{p}_{\text{Cart}} = \mathbf{p}_0 \wedge \mathbf{r}_{\text{Cart}} = \mathbf{r}_0$ , and the invariant is  $\mathbf{p}_{\text{Cart}}(t) = \mathbf{p}_0 \wedge \mathbf{r}_{\text{Cart}}(t) = \mathbf{r}_0 \wedge \mathbf{s}_{\text{Cart}} = 0$   
 607 for all  $t \in [0, T]$ .

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

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

## 616 5.2 Motion Calculus with Guarded Choice

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

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

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

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

630 The subtyping and typing relation are updated as follows:

$$\begin{array}{c}
 \text{[SUB-OUT]} \\
 \forall i \in I : \quad S_i \leq S'_i \quad T_i \leq T'_i \quad \mathcal{P}_i \Rightarrow \mathcal{P}'_i \\
 \hline
 \oplus\{[\mathcal{P}_i] \mathbf{p}! \ell_i(S_i).T_i\}_{i \in I} \leq \oplus\{[\mathcal{P}'_i] \mathbf{p}! \ell_i(S'_i).T'_i\}_{i \in I \cup J}
 \end{array}$$

631

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

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

636

637 Type checking the rules propagates the expression from **if then else** and matches it into a  
 638 branch of the type. To deal with the **else** branch we modify the predicate in the remaining

639 branches of the type. For the last else branch of a, possibly nested, if then else we need  
640 the following extra rule:

$$641 \frac{[\text{T-CHOICE-FINAL}]}{\frac{\Gamma \vdash P : T}{\Gamma \vdash P : \oplus\{\{\text{true}\}T\}}}$$

642 ► **Example 5.2.** Usually, for the propagation of tested expressions through the branches we  
643 modify the type. Let us make an example of how this works. Consider we have the following  
644 process if  $e_1$  then  $P_1$  else  $P_2$  which has the type  $\oplus\{\{e_1\}T_1, \{\neg e_1\}T_2\}$ . Assuming that  $P_i : T_i$   
645 for  $i \in \{1, 2\}$  we can build the following derivation:

$$646 \frac{e_1 \Rightarrow e_1 \quad \Gamma \vdash P_1 : T_1 \quad \frac{\Gamma \vdash P_2 : T_2}{\Gamma \vdash P_2 : \oplus\{\{e_1 \vee \neg e_1\}T_2\}}}{\Gamma \vdash \text{if } e_1 \text{ then } P_1 \text{ else } P_2 : \oplus\{\{e_1\}T_1, \{\neg e_1\}T_2\}}$$

647 With a bit of boolean algebra, we can show that  $e_1 \vee \neg e_1 \Leftrightarrow \text{true}$ .

### 648 5.3 Existence of Joint Trajectories and Verification

649 The goal of the compatibility check is to make sure that abstract motion primitives specified in  
650 a global type can execute concurrently. This requires two checks. First, for motion primitives  
651 of different processes executed in parallel, we need to make sure that there exists a trajectory  
652 satisfying all the constraints of the motion primitives. Second, for motion primitives executed  
653 sequentially by the same process, we need to make sure that the post-condition of the first  
654 implies the pre-condition of the second motion primitive, taking into account the guards of  
655 choices in the middle.

656 To check that motion primitives executing in parallel have a joint trajectory, we use an  
657 assume-guarantee style of reasoning. When two processes are attached, one process relies  
658 on the invariants of the other's output (which can be an external input) to satisfy its own  
659 invariant and vice versa. We refer to standard methods [35, 4] for the details.

660 For the allowed trajectories, we need to also check the absence of collision. This means  
661 that once we have the constraints defining a joint trajectory  $\xi$  to check that for any two  
662 distinct processes  $p$  and  $q$  the property  $rsrc_p(\xi_p) \cap rsrc_q(\xi_q) = \emptyset$ .

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

## 669 6 Evaluation

### 670 6.1 Implementation

671 We have implemented the system we describe on top of PGCD [4]<sup>1</sup>, a system for programming  
672 and verification of robotic systems. PGCD is build on top of the Robotic Operating System

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

673 (ROS) [40], a software ecosystem for robots. The core of ROS is a publish-subscribe messaging  
674 system. PGCD uses ROS’s messaging to implement its synchronous message-passing layer.  
675 On the verification side, PGCD uses a mix of model-checking (using SPIN [22]) to deal with  
676 the message-passing structure, and symbolic reasoning (using SYMPY [33]) and constraint  
677 solving (using DREAL [16]) to reason about motion primitives.

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

687 The typing, including subtyping, is implemented by computing an alternating simulation  
688 [2] between programs and their respective local type. Intuitively, an alternative refinement  
689 relation check that a process implements its specification (subset of the behaviours) without  
690 restricting the other processes. For synchronous message passing programs, the subtyping  
691 relation for session type matches alternating refinement. We use this view on subtyping as  
692 the theory of alternating simulation [2] gives us an algorithm to compute this relation and,  
693 therefore, check subtyping.

## 694 6.2 Experiments

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

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

### 700 Setup

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

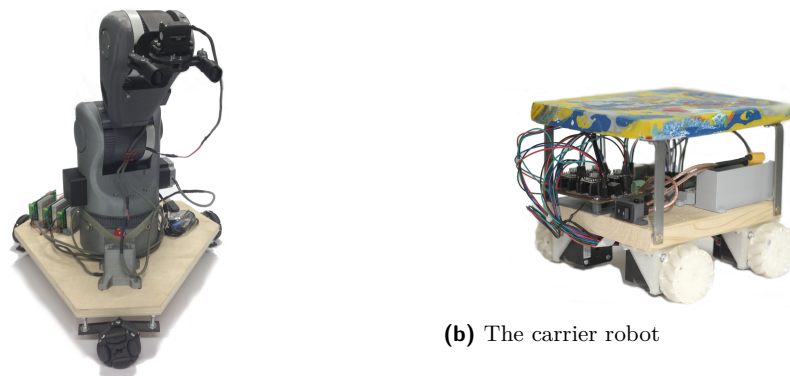
703 **Arm** The arm is a modified BCN3D MOVEO,<sup>2</sup> where the upper arm section is shortened to  
704 make it lighter and easier to mount on the cart. The arm with its control electronics is  
705 mounted on top of the cart.

706 **Cart** The cart is shown on Figure 4a. The control electronics and motors are situated below  
707 the wooden board. The cart is an omnidirectional driving platform. It uses omniwheels to  
708 get three degrees of freedom (two in translation, one in rotation) and can move between  
709 any two positions on a flat ground. The advantage of using such wheels is that all  
710 the three degrees of freedom are controllable and movement does not require complex  
711 planning. Due to the large power consumption of the arm mounted on top, this cart is  
712 powered by a tether.

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

---

<sup>2</sup> <https://github.com/BCN3D/BCN3D-Moveo>



(a) The cart and arm robots attached together

(b) The carrier robot

■ **Figure 4** Robots used in our experiments

715 All the three robots use stepper motors to move precisely. The robots do not have  
 716 feedback on their position and keep track of their state using *dead reckoning*, i.e., they know  
 717 their initial state and then they update their virtual state by counting the number of steps  
 718 the motors turns. If we control slippage and do not exceed the maximum torque of the  
 719 motors, there is little accumulation of error as long as the initial state is known accurately.  
 720 In our experiments, we use markings on the ground to fix the initial state as can be seen in  
 721 Figure 5. Furthermore, using stepper motors allows us to know the time it takes to execute  
 722 a given motion primitive by fixing the rate of steps.

723 Each robot has a RaspberryPi 3 model B to run the program. The ROS master node,  
 724 providing core messaging services, runs on a separate laptop to which all the robots connect.  
 725 The RaspberryPi runs Raspbian OS (based on Debian Jessie) and the laptop runs Ubuntu  
 726 16.04. The ROS version is Kinetic Kame.

## 727 Experiments

728 We describe two experiments:

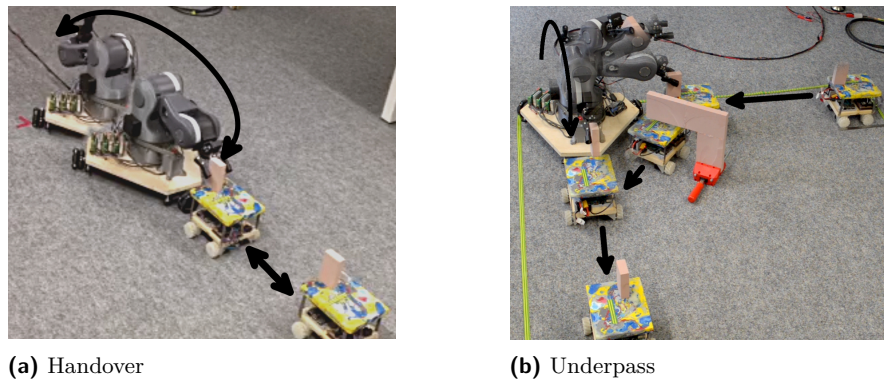
729 **Handover.** This experiment corresponds to our earlier example. The two carts meet before  
 730 the arm takes an object placed on top of the carrier and, then, they go back to their  
 731 initial position (see Figure 5a).

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

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

743 Table 3 shows the size of the programs in the language of PGCD (sum for all the robots)  
 744 and the size of the global specifications. As part of the program we include a description of  
 745 the environment which specifies the initial states of the robots and the obstacles used for  
 746 additional collision checks. Finally, we show the number of verification conditions (#VCs)





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

■ **Table 3** Programs, Specification, and Checks

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

747 generated during the subtyping and the checks for joint trajectories. The total running time  
 748 includes all the steps, i.e., checking the global specification, projection, typing, the existence  
 749 of joint trajectories, and the absence of collision. The running time is dominated by the check  
 750 on trajectories and collisions. The motion primitives (implementation and specification) are  
 751 taken from PGCD without any change and represent around 1K lines of codes for all three  
 752 robots.

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

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

## 760 **7 Related Work**

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

769 At the specification level, hybrid process algebras and other models of hybrid systems  
 770 [1, 41, 7, 38] can model concurrent hybrid systems. However, these papers do not provide a  
 771 direct path to implementation. Hybrid extensions to synchronous reactive languages [6, 5]  
 772 describe programs which interact through events and control physical variables. Most existing  
 773 verification methodologies for these programs rely on global model checking rather than on  
 774 types. Our choice of session types is inspired by efficient type checking but also as the basis  
 775 for describing interface specifications for components.

776 Extensions and applications of multiparty session types have been proposed in many  
 777 different settings. See, e.g. [27, 3, 17]. We discuss only most related work. The work [9]  
 778 extends multiparty session types with time, to enable the verification of realtime distributed  
 779 systems. This extension with time allows specifications to express properties on the causalities  
 780 of interactions, on the carried data types, and on the times in which interactions occur. The  
 781 projected local types correspond to Communicating Timed Automata (CTA). To ensure the  
 782 progress and liveness properties for projected local types, the framework requires several  
 783 additional constraints on the shape of global protocols, such as feasibility condition (at  
 784 any point of the protocol the current time constraint should be satisfiable for any possible  
 785 past) and a limitation to the recursion where in the loop, the clock should be always reset.  
 786 The approach is implemented in Python in [34] for runtime monitoring for the distributed  
 787 system. Later, the work in [8] develops more relaxed conditions in CTAs, and applies them to  
 788 synthesise timed global protocols. Unlike our work, no type checking for processes is studied  
 789 in [8]. The main difference from [9, 34, 8] is that our approach does not rely on CTAs and is  
 790 more specific to robotics applications where the verification is divided into the two layers; (1)  
 791 a simple type check for processes with motion primitives to ensure communication deadlock-  
 792 freedom with global synchronisations; and (2) additional more refined checks for trajectories  
 793 and resources. This two layered approach considerably simplifies our core calculus and typing  
 794 system in Section 4, allowing to verify more complex scenarios for robotics interactions.

## 795 **8 Conclusion**

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

804 At this point, our language is a starting point and not a panacea for robotics programming.  
 805 Decoupling specifications into parallel and/or sequential tasks and using distributed controllers  
 806 assumes “loosely coupled dynamics.” In some examples, such as a multiple cart/arm co-  
 807 ordination control, it may not be easy to assume a purely distributed control strategy based on  
 808 independent motion primitives. We are thus exploring simultaneous concurrent programming  
 809 and distributed controller and co-ordinator synthesis. As an example, assume that we have  
 810 two cart/arm compositions which should lift one object together. In particular we can assume  
 811 that lifting the object with only one arm would cause the cart/arm compositions to tilt over,  
 812 which generates a strong coupling between all components during the coordinated lift of the  
 813 object. Our framework allows to easily synchronise all the components. However, in any  
 814 realistic scenario a robust controller would need (almost) continuous feedback between all

815 components to fulfill the coordinated lift task. Thus, our model of loosely coupled motion  
 816 primitives, one per component, may be too weak or incur too much communication and  
 817 bandwidth overhead for a real implementation.

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

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

### 830 ——— References ———

- 831 **1** R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR '97: Concurrency Theory*, volume 1243 of *LNCS*, pages 74–88. Springer, 1997.
- 832 **2** R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR'98 Concurrency Theory*, pages 163–178. Springer, 1998.
- 833 **3** Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-  
 834 Malo Denielou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch  
 835 Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nich-  
 836 olas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in  
 837 Programming Languages. *FTPL*, 3(2-3):95–230, 2016.
- 838 **4** Gregor B. Banusic, Rupak Majumdar, Marcus Pirron, Anne-Kathrin Schmuck, and Damien  
 839 Zufferey. PGCD: robot programming and verification with geometry, concurrency, and  
 840 dynamics. In Xue Liu, Paulo Tabuada, Miroslav Pajic, and Linda Bushnell, editors, *Proceedings*  
 841 *of the 10th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2019,*  
 842 *Montreal, QC, Canada, April 16-18, 2019*, pages 57–66. ACM, 2019. doi:10.1145/3302509.  
 843 3311052.
- 844 **5** Kerstin Bauer and Klaus Schneider. From synchronous programs to symbolic representations of  
 845 hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems:*  
 846 *Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 41–50.  
 847 ACM, 2010. doi:10.1145/1755952.1755960.
- 848 **6** Albert Benveniste, Timothy Bourke, Benoît Caillaud, Jean-Louis Colaço, Cédric Pasteur,  
 849 and Marc Pouzet. Building a hybrid systems modeler on synchronous languages principles. *Proceedings of the IEEE*, 106(9):1568–1592, 2018. doi:10.1109/JPROC.2018.2858016.
- 850 **7** J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer*  
 851 *Science*, 335(2):215 – 280, 2005. Process Algebra. doi:https://doi.org/10.1016/j.tcs.  
 852 2004.04.019.
- 853 **8** Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting Deadlines Together. In *26th*  
 854 *International Conference on Concurrency Theory*, volume 42 of *LIPICs*, pages 283–296. Schloss  
 855 Dagstuhl, 2015.
- 856 **9** Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed Multiparty Session Types. In  
 857 *25th International Conference on Concurrency Theory*, volume 8704 of *LNCS*, pages 419–434.  
 858 Springer, 2014.

- 862 **10** Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle  
863 introduction to multiparty asynchronous session types. In *SFM*, volume 9104 of *LNCS*, pages  
864 146–178. Springer, 2015.
- 865 **11** Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty session types meet communicating  
866 automata. In *ESOP 2012 - European Symposium on Programming*. Springer, 2012. doi:  
867 10.1007/978-3-642-28869-2\_10.
- 868 **12** Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised  
869 multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/  
870 LMCS-8(4:6)2012.
- 871 **13** Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised  
872 multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/  
873 LMCS-8(4:6)2012.
- 874 **14** Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and  
875 Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. In *PLACES*, volume  
876 203 of *EPTCS*, pages 29–43, 2015. doi:10.4204/EPTCS.203.3.
- 877 **15** Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and  
878 Nobuko Yoshida. Denotational and operational preciseness of subtyping: A roadmap. In  
879 *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion*  
880 *of His 60th Birthday*, volume 9660 of *LNCS*, pages 155–172. Springer, 2016.
- 881 **16** Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear  
882 theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 -*  
883 *24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14,*  
884 *2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer,  
885 2013. doi:10.1007/978-3-642-38574-2\_14.
- 886 **17** Simon Gay and Antonio Ravera, editors. *Behavioural Types: from Theory to Tools*. River  
887 Publishers, 2017.
- 888 **18** Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida.  
889 Precise subtyping for synchronous multiparty sessions. *J. Log. Algebr. Meth. Program.*,  
890 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.
- 891 **19** Sehoon Ha, Stelian Coros, Alexander Alspach, James M. Bern, Joohyung Kim, and Katsu  
892 Yamane. Computational design of robotic devices from high-level motion specifications. *IEEE*  
893 *Trans. Robotics*, 34(5):1240–1251, 2018. doi:10.1109/TR0.2018.2830419.
- 894 **20** Thomas A. Henzinger. Sooner is safer than later. *Inf. Process. Lett.*, 43(3):135–141, 1992.  
895 doi:10.1016/0020-0190(92)90005-G.
- 896 **21** Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE*  
897 *Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30,*  
898 *1996*, pages 278–292. IEEE Computer Society, 1996. doi:10.1109/LICS.1996.561342.
- 899 **22** G.J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.  
900 doi:10.1109/32.588521.
- 901 **23** Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, pages 509–523, 1993.
- 902 **24** Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type  
903 disciplines for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*,  
904 pages 22–138. Springer, 1998. doi:10.1007/BFb0053567.
- 905 **25** Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types.  
906 In *POPL*, pages 273–284. ACM Press, 2008. doi:10.1145/1328438.1328472.
- 907 **26** Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types.  
908 *Journal of ACM*, 63:1–67, 2016.
- 909 **27** Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo  
910 Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres  
911 Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM*  
912 *Comput. Surv.*, 49(1), 2016. doi:10.1145/2873052.

- 913 28 R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris:  
914 Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL 15*, pages  
915 637–650. ACM, 2015.
- 916 29 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. In Pedro R.  
917 D’Argenio and Hernán C. Melgratti, editors, *CONCUR*, volume 8052 of *LNCS*, pages 395–409.  
918 Springer, 2013. doi:10.1145/1328438.1328472.
- 919 30 Dimitrios Kouzapas and Nobuko Yoshida. Globally governed session semantics. *Logical*  
920 *Methods in Computer Science*, 10(4), 2015.
- 921 31 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical  
922 choreographies. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*  
923 *Languages*, pages 221–232. ACM, 2015.
- 924 32 A.M. Mehta, N. Bezzo, P. Gebhard, B. An, V. Kumar, I. Lee, and D. Rus. A design environment  
925 for the rapid specification and fabrication of printable robots. *Experimental Robotics*, pages  
926 435–449, 2015.
- 927 33 Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev,  
928 Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina  
929 Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta,  
930 Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán  
931 Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony  
932 Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, January  
933 2017. doi:10.7717/peerj-cs.103.
- 934 34 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for  
935 multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.
- 936 35 Pierluigi Nuzzo. *Compositional Design of Cyber-Physical Systems Using Contracts*. PhD thesis,  
937 EECS Department, University of California, Berkeley, Aug 2015. URL: [http://www2.eecs.  
938 berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html](http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-189.html).
- 939 36 P.W. O’Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-  
940 3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.
- 941 37 Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- 942 38 A. Platzer. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*.  
943 Springer, 2010.
- 944 39 K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2-3):285–327,  
945 1995. doi:10.1016/0167-6423(95)00017-8.
- 946 40 Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob  
947 Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop*  
948 *on open source software*, 2009.
- 949 41 W.C. Rounds and H. Song. The phi-calculus: A language for distributed control of reconfigur-  
950 able embedded systems. In *HSCC*, pages 435–449. Springer, 2003.
- 951 42 Alceste Scalas and Nobuko Yoshida. Less is more: Multiparty session types revisited. *Proc.*  
952 *ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.
- 953 43 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its  
954 Typing System. In *PARLE’94*, volume 817 of *LNCS*, pages 398–413, 1994. doi:10.1007/  
955 3-540-58184-7\_118.