Trustworthy Pervasive Healthcare Services via Multiparty Session Types

Anders S. Henriksen[1], Lasse Nielsen[1], Thomas T. Hildebrandt[2], Nobuko Yoshida[3], and Fritz Henglein[1]

[1] University of Copenhagen
[2] IT University of Copenhagen
[3] Imperial College London

Abstract This paper proposes a new theory of multiparty session types with assertions based on symmetric sum types and demonstrates its applicability to collaborative workflows in healthcare systems for clinical practice guidelines (CPGs). The theory leads to a model-driven implementation of a prototype tool for CPGs which automatically generates deadlock-free distributed programs from user-friendly declarative workflows specified as a Process Matrix spreadsheet. In addition to safety properties, the generated code can ensure the logical properties declared in a Process Matrix. They are subsequently interpreted to provide a trustworthy pervasive workflow execution on Android tablet PCs. We also report on a demonstration of the prototype to a physician, who after having seen an example healthcare workflow being executed, was able to specify her own healthcare workflow declaratively as a Process Matrix spreadsheet and immediately test it on the Android tablet PCs.

1 Introduction

Healthcare processes are characterised by being highly mobile, collaborative, security critical, and requiring a high-degree of flexibility and adaptability. Hereto comes that they typically involve complex decisions based on data collected during the process and are regulated, e.g. by law and clinical practice guidelines (CPGs) [6]. These characteristics make healthcare processes a particular challenging example of case management processes [**?**] in need for computerised support based on formalised and verifiable process models.

CPGs are descriptions of medical treatment procedures, practised globally with local variations, in order to treat specific medical disorders. CPGs can express workflows and various cooperations among healthcare processes which are formed by the diverse collaborative patterns between organisations. That is, a CPG is an agreement of *global protocol* or *guideline* between distributed multiple organisations or participants. A pattern, which plays a prominent role in CPGs, is what we will call *symmetric, multiparty synchronisation* where the participants collectively decide on one of the possible choices of possible next steps in the protocol.

Such global protocols with symmetric, multiparty synchronizations are naturally expressed in a *choreography language*, such as the WS-CDL [8] or the recent BPMN 2.0 Choreography diagram notation [] exemplified in Fig. 1 in § 2. While workflow models, and also CPGs, traditionally have been represented as flow-graphs inspired by and based on the seminal work on the Petri Nets model [] and verified using model checking techniques, workflows represented as choreographies offer an alternative approach to modeling and verification of distributed workflows, and CPGs in particular, based on *type checking*. Put shortly, the work on session types and end-point projections [2] provides a foundation for adding behavioral types to choreographies and project the choreographies to typed end-point processes (i.e. corresponding to BPMN processes) that are guaranteed to be deadlock-free.

Within the last five years, a number of researchers have pointed out that imperative flow-graph models such as BPMN processes and choreographies have limitations when it comes to flexibility and adaptability [**?**]. As an alternative, formal declarative process notations have been proposed and investigated as a means to provide more support for adaptability in case management systems in general [7,**?**,**?**] and health care processes in particular [**?**] and verified, e.g. using the logic itself [] or model checking techniques [].

In the present paper we show how the theory of multiparty session types [2] extended with logical predicates [1] and symmetric sum types [3] can be used to represent declarative, distributed, collaborative workflows, that can be modeled declaratively by domain experts and verified statically, i.e. at compile time, using automatic code generation and type inference.

Concretely we show how collaborative healthcare workflows declared as *Process Matrix spreadsheets* can be automatically mapped to session typed distributed programs which are interpreted to provide a trustworthy pervasive workflow execution on Android tablet PCs. We then report on a demonstration of the prototype to a physician, who after having seen an example healthcare workflow being executed, was able to specify her own healthcare workflow declaratively as a Process Matrix spreadsheet and immediately test it on the Android tablet PCs.

2 From Spreadsheets via Types to Pervasive Services

In this section we give an overview of the prototype implementation and the different technologies used by means of a simple example workflow. First in § 2.1 we describe the example workflow as a BPMN 2.0 Choreography diagram and the corresponding Process Matrix spreadsheet.

2.1 Example Workflow as Choreography and Process Matrix

A simple CPG workflow involving three participants is described in Fig. 1 as a Choreograpy diagram in the Business Process Modelling Notation (BPMN) 2.0. The described workflow is activated, when a patient is admitted (indicated by the start event). Then two tests are executed in parallel by a nurse. Note that each activity box is a *communication* between the three participants with one initiator (indicated in the white ribbon) and two receivers (indicated in the shaded ribbons). Thus, the test results are send by the nurse to the patient and the doctor. Each test may be repeated, as signaled by the repeating sub-process arrow, e.g. if the test failed or the result was not clear. Then, depending on the results of the tests, either the patient is discharged directly, or the doctor prescribes a drug to the patient

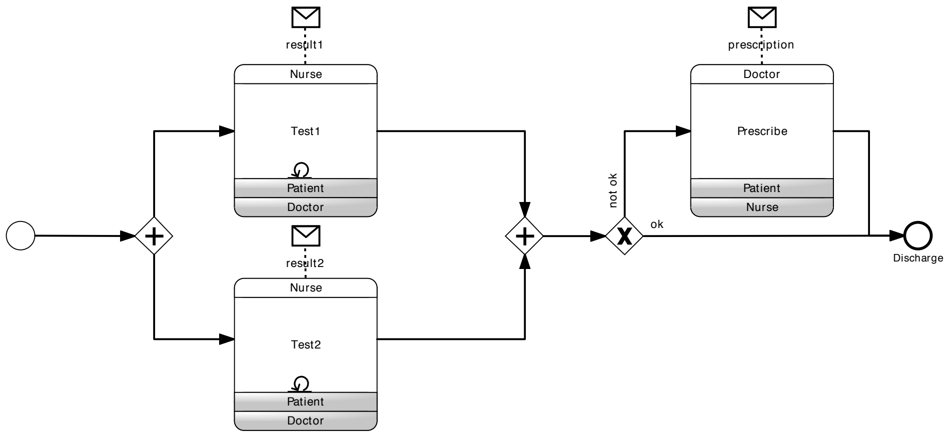


Figure 1. Workflow as BPMN 2.0 Choreography

before discharging, sending the prescription to both the patient and the nurse. The workflow is ended, when the patient is discharged. The described workflow is a standard paradigm in CPGs, that is, first a set of tests are performed, and depending on the results, either more tests are performed, the patient is discharged or a treatment is executed. In this workflow the treatment consists of simply prescribing a drug to the patient.

For our demonstrator we do not use BPMN 2.0 choreography diagrams. Instead we use a simplified version of the declarative *Process Matrix* representation developed by our industrial partner Resultmaker in the TrustCare research project. The process matrix corresponding to the choreography in Fig. 1 is shown in Fig. 2 below.

Id	Name	P	D	N	Seq	Log	Condition	Input	Action
1.1.1	Test1	R	R	W			$\neg$ pre	result1	
1.1.2	Test2	R	R	W			$\neg$ pre	result2	
1.2.1	Prescribe	R	W	R	1.1.1, 1.1.2		$\neg$ pre $\wedge$ $\neg$ (result1 $\wedge$ result2)		set(pre)
1.3.1	Discharge	R	W	R	1.1.1, 1.1.2		(result1 $\wedge$ result2) $\vee$ pre		end

Figure 2. Example CPG workflow as Process Matrix.

The process matrix has a row for each activity, and columns providing a (unique) Id, a name, access control (**R**ead or **W**rite) for each participant (**P**atient, **N**urse, **D**octor), the **Seq**uential predecessor relation, the **Log**ical predecessor relation (not used in our simple example), activity Conditions (based on data), Input data, and a possible Action performed when the activity is executed (e.g. **set**ting a boolean data field or **end**ing the workflow). In more detail, the Action column contains either the **end** command or a command given in a small if-then-else language:

$$\text{Cmd} ::= \varepsilon \mid \text{set}(x) \mid \text{reset}(x) \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \{c_1; \ldots; c_n\}.$$

ε command means no action, *set* and *reset* sets the value of the variable to true or false respectively. If-then-else considers a boolean expression and then uses either of the commands. The bracketed commands are performed in sequence.

The condition field must evaluate to true for an activity to be enabled. Further, if an activity, like Prescribe or Discharge in our example, has sequential predecessors, every predecessor activity (for which the condition field presently evaluates to true) must have been executed at least once before the activity can be executed. This means that by default an activity with no sequential or logical predecessors and no conditions can be executed at any time and any number of times. In other words, flexibility is the default, if the flow should be constrained the constraints must be explicitly given. For instance, if tests should be allowed also after a prescription, possibly leading to a new prescription, one could change the matrix to the one given in Fig. 3 below.

Id	Name	P	D	N	Seq	Log	Condition	Input	Action
1.1.1	Test1	R	R	W				result1	reset(pre)
1.1.2	Test2	R	R	W				result2	reset(pre)
1.2.1	Prescribe	R	W	R	1.1.1, 1.1.2		$\neg$ pre $\wedge$ $\neg$ (result1 $\wedge$ result2)		set(pre)
1.3.1	Discharge	R	W	R	1.1.1, 1.1.2		(result1 $\wedge$ result2) $\vee$ pre		end

Figure 3. Example slightly more flexible CPG workflow as Process Matrix.

Besides being easier to implement (it just required an off-the-shelf spreadsheet program and parsing the standard output format), the notation thus also gives some extra possibilities for flexiblility in adaptation and execution of the workflow. These possibilities show more clearly in the experiment we did with a doctor creating her own CPG Process Matrix spreadsheet as described in § 3.

2.2 Example Workflow as Multiparty Session Type

We now demonstrate how process matrix workflow processes as given above can be described compactly in multiparty session types with logical predicates as assertions and so-called symmetric sum types.

Multiparty session types [2] define protocols for interactions in a group of participants which are guaranteed progress, and correspond in fact closely to choreographies. In addition to defining the protocol, the theory of session types also allows to verify that a collection of π-calculus processes, corresponding to BPMN processes in a collaboration diagram describing each participant, follow the specified protocol. The extension of multiparty session types with *assertions* [1] elaborates type signatures through logical predicates, which can be used to restrict the values that are communicated and choices that are made. We also use *symmetric sum types* [4] which is an extension of the multiparty session types that can type nondeterministic choice agreed upon between several participants.

These three main features, multiparty, symmetric synchronisations and logical predicates are all essential to represent process matrix workflows in a direct and compact way, and verify practical usecases, not only in the context of CPGs, but also for workflows in general.

```
μ   workflow⟨test1:Bool=false, test2:Bool=false, pre:Bool=false,
              result1:Bool=false, result2:Bool=false⟩ .
{ Test1 [[not pre]]:
   3→1:1⟨Bool⟩ as x;          // The result of test1
   3→2:2⟨Bool⟩ as y[[x=y]];  // The result of test1
    workflow⟨true, test2, pre, x, result2⟩,
  Test2 [[not pre]]:
   3→1:1⟨Bool⟩ as x;          // The result of test2
   3→2:2⟨Bool⟩ as y[[x=y]];  // The result of test2
    workflow⟨test1, true, pre, result1, x⟩,
 Prescribe [[test1 and test2 and not pre and not (result1 and result2)]]:
   2→1:3⟨String⟩;             // The prescription
   2→3:4⟨String⟩;             // The prescription
    workflow⟨test1, test2, true, result1, result2⟩,
 Discharge [[test1 and test2 and ((result1 and result2) or pre)]]:
   end
}
```

Figure 4. Session type representation of workflow using assertions

Fig. 4 specifies the workflow from Fig. 1 as a multiparty session type with symmetric sum types and assertions.

The workflow is described by a *recursive type* (indicated by the initial μ sign), parametrised by a *state*: test1 and test2 describe if the respective test actions have been executed, which is needed because the test actions are sequential predecessors of the prescribe and discharge actions. pre is the condition recording if the prescription activity has been executed, used to restrict it to be only executed once and blocking subsequent test1 and test2 actions. Finally, result1 and result2 record the results of the respective tests. After executing an action, the recursive type is used with an updated state, except if the action like Discharge is an end action. In the state where both tests have been executed, no prescription has been made and not both tests where ok (represented as the boolean value true), the Prescribe action is enabled.

The specification also describes that when Test1 is executed, the result is sent from participant 3 (the nurse) to participant 1 (the patient) and 2 (the doctor) (represented by $3 \to 1$ and $3 \to 2$).

The logical assertions are also useful for other aspects of the CPG workflows. Assertions can for example be used to ensure that the prescribed doses of medicine are below the lethal limit, however this has not been done in our example. In the example workflow the assertions are also used to ensure, that the result sent to the patient and the doctor after a test is the same. The results of the tests are then used to indicate if it makes sense to administer the medicine or discharge the patient directly.

2.3 Implementation

The demonstrator allows distributed execution of process matrix specified workflows on Android tablets. The architecture/use-case is depicted in Fig. 5 below.

The arrow from the CPG cloud indicates that the process designer describes a workflow (e.g. a CPG) as a process matrix specification in a spreadsheet. The arrow to the mps (multi-party session types) code generator indicates that it takes the process matrix as input. In the demonstrator implementation, the process matrix is given as a comma-separated file produced as a standard output from

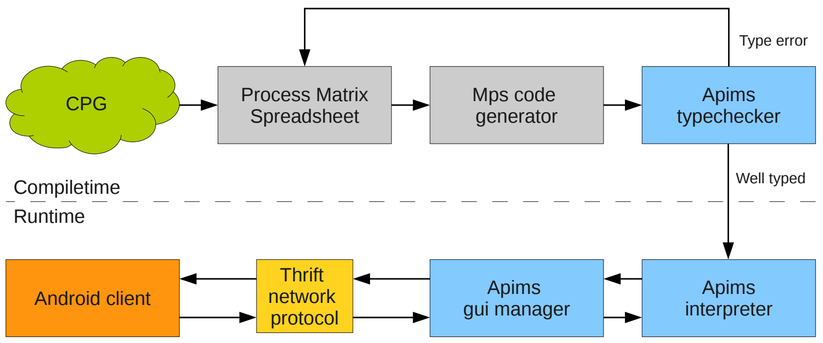


Figure 5. Demonstrator architecture.

an off-the-shelf spreadsheet program. Thus, it enables the process matrix to be specified in a normal spreadsheet program, that provides a graphical editor, that is familiar to many end-users, "for free".

The generated mps code consists of a global multi-party session type, as exemplified in Fig. 4, representing the entire workflow, and end-point π-calculus (with plenum synchronization allowing to implement the symmetric sum types) terms for each participant.

The generated code is augmented with user-interface information generated from descriptions written in a separate spreadsheet table. In Fig. 6 we show code for the process matrix from Fig. 2, that is very close to the actual generated code. We only show the Doctor part, as the global type is very similar to the one shown in Fig. 4. (The number 2 appearing in the code several places indicate that this is participant 2, the doctor).

Corresponding to the recursion in the global session-type in Fig. 4, the generated mps-code consists of a single loop (line 5-31), where all actions specified in the matrix correspond to a branch (lines 9, 14, 19, 26) in a single synchronisation. Each branch is annotated with the writer of that action. Differently from BPMN Choreographies, a process matrix allows actions with more that one writer. This will be compiled to several branches in the synchronisation (e.g. if the nurse could also discharge the patient, there would be a branch `a131-n`).

The loop maintains a state, which includes the executed state of each action, and each logical variable. Each writer action receives inputs from the gui (line 20) and sends them to the reader participants (lines 21, 22).

Predecessors and the activity condition are enforced using the state. Using an assertion for each branch, we can make sure a branch is only shown when its predecessors have been executed and the activity condition is true. When looping in the end of each branch, the executed variables are updated in two ways:

- The executed state of the completed action, is set to true (e.g. a111 in line 12).
- The executed state of any actions that have the completed action as logical predecessor, is set to false.

```
1   link(3, wf, s, 2);
2   guivalue(3, s, 2, ".uid", "d");
3   guivalue(3, s, 2, "a121_d:title", "Prescribe");
4   ...
5   def Loop⟨a111: Bool, a112: Bool, a121: Bool, a131: Bool,
6            res1: Bool, res2: Bool, pre: Bool⟩
7           (w:  wf⟨a111, a112, a121, a131, res1, res2, pre⟩@(2 of 3)) =
8   guisync(3, w, 2) {
9   a111-n 1[[not pre]]():
10    w[7] ? lungs_ok;
11    guivalue(3, w, 2, "Lungs ok?:info", lungs_ok);
12    Loop⟨true, a112, a121, a131,
13         (lungs_ok or ((not lungs_ok) and res1)), res2, pre⟩(w),
14  a112-n 1[[not pre]]():
15    w[7] ? throat_ok;
16    guivalue(3, w, 2, "Throat ok?:info", throat_ok);
17    Loop⟨a111, true, a121, a131, res1,
18        (throat_ok or ((not throat_ok) and res2)), pre⟩(w),
19  a121-d [[a111 and a112 and ((not pre) and (not (res1 and res2)))]]
20         (prescription: String = ""):
21    w[3] ! prescription;
22    w[5] ! prescription;
23    guivalue(3, w, 2, "Prescription:info", prescription);
24    guivalue(3, w, 2, ".a121_d", true);
25    Loop⟨a111, a112, true, a131, res1, res2, true⟩(w),
26  a131-d [[a111 and a112 and ((res1 and res2) or pre)]]
27         (dis_comment: String = ""):
28    end
29  }
30  in
31  Loop⟨false, false, false, false, false, false, false⟩(s)
```

Figure 6. Mps code for Doctor participant.

The last part of the logic is the extra control column. The effect of the set command for action a121 can be seen in line 25, where the variable *pre* is set to true.

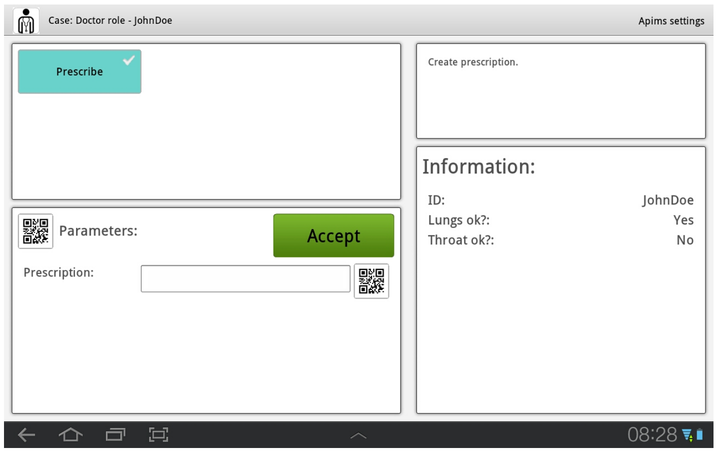
The next arrow in Fig. 5 shows that the mps code is type checked with the Apims type checker. If the code is not well-typed it will in this case be because the workflow can reach a deadlock state, i.e. a state which is not the final state, but no activity can be executed. The example process matrices given above produces well-typed code, however, an innocently looking modification changing the logical or ($\vee$) in the condition for the Discharge activity to a logical and ($\wedge$) would make it possible to deadlock if both tests are fine (blocking the prescription), which is now required in order to discharge the patient. The static type checking thus allow the designer, at Compile time, to catch potential deadlocks before the workflow is initiated and return to the spreadsheet and revise the specification as indicated by the arrow back to the Process Matrix Spreadsheet. (Of course the error may also be in the design of the non-formal CPG description which served as input to the design of the process matrix).

When the code is well-typed, the Apims interpreter can execute the code of each participant process, controlling the user interface of the clients (e.g. tablet pcs, mobile phones or computers), as indicated to the right in the Run time part of Fig. 5

Besides the core part, maintaining the workflow state and executing each step as described above, the Apims interpreter consists of a gui manager (linked in as a separate, replaceable module) that communicates with the clients and maintains a viewpoint for each participant. In more detail, each guisync term introduces a list of *choices* for each participant, corrsponding to enabled branches in the workflow, and each guivalue a list of *values* for each participant. The gui manager maintains data structures for these two components, and the clients interact with the workflow by manipulating these components. The choices can either be accepted, and if all participants accept a choice the execution can continue with the corresponding branch.

The values are used to send data to the client. There are several kind of data being transmitted: meta data, e.g. the human readable name of the different actions (and specified in the spreadsheet); value data, e.g. the data entered by the other participants, execution data, e.g. the execution state of each action.

All these different pieces of data are encoded in the key-value pair of each guivalue. By keeping all data in the values maintained by the interpreter, no information is kept on the clients and clients can therefore be changed/break down without ruining the execution.



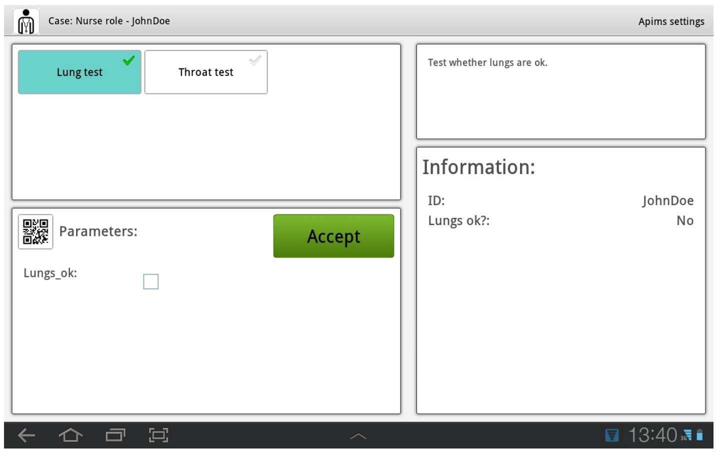
Information:

ID:	JohnDoe
Lungs ok?:	Yes
Throat ok?:	No

Figure 7. Screenshot from Android client.

In the example the guivalue in line 2 assigns the Doctor role to the specific part of the code, this allows the gui manager to know which choices are assigned to which role. Fig. 7 shows a screenshot of the Android client running the example workflow as the doctor role, which can be seen in top of the screen. The workflow is in a state where the nurse has performed the two tests. The other guivalues all correspond to different parts of the screen. The guivalue in line 3, forms parts of the meta data, and assigns the human-readable name “Prescribe” to the action `a121-d` . The guivalues in lines 11 and 16 are used to show the information received from the nurse (the result of the tests), which can be seen in the window to the right. Similarly the guivalue in lines 23 are used to show the values to the right when the Doctor has entered those. The last guivalue in lines 24 are used to pass the execution state of the action to the client. This is used to show a small green checkmark for the action, so the user know that it has been performed. In the screenshot the execution state is false, so the checkmark is not green.

It is important to stress that every participant uses the same generic Android client. The gui manager uses the generated code to make sure that the Android client used by the Doctor presents the ”end-point” process of the workflow relevant for the doctor and the Android client used by the Nurse presents the ”end-point” relevant for the Nurse. An example screenshot of the Android client running the example as the nurse role is shown in Fig. 8. It shows the workflow in a state where the nurse has performed the Lung test, with a negative result, and still needs to perform the throat test.


Case: Nurse role - JohnDoe

Apims settings

Lung test ✓

Throat test ✓

Test whether lungs are ok.

Information:

ID:	JohnDoe
Lungs ok?:	No

Parameters:

Accept

Lungs_ok: ☐

13:40

Figure 8. Screenshot from Android client.

The communication protocol used relies on Thrift [5] which enables clients written in other languages to be used together with the Android client already developed.

When developing for tablet clients there are, in addition to the technical challenges, user interface challenges, e.g. it is harder to input large pieces of text using a tablet keyboard instead of a regular keyboard. We have only touched on these challenges, but a simple, yet quite beneficial approach has been to allow input using qr-codes scanned though the tablets camera. This enables a user to scan the drug name and the dose from the physical objects, minimising the amount of typing needed.

3 Experiment: An End-user Developed Workflow

To test the developed software, we performed a simple experiment with the help of a physician: Dorthe Furstrand Lauritzen (DFL). The motivation behind the experiment was to get first hand impressions from a domain expert, to evaluate the current implementation and set goals for future development. Although DFL is a physician and not a computer scientist, she has experience with use of it and in particular implementation of CPGs. However, she had never seen any of the techniques used in the demonstrator before, in particular the declarative process matrix notation was completely new to her.

The main component of the experiment was to put DFL in the role of the workflow designer, letting her use the spreadsheets to formalise a simple self-chosen medical workflow, which can be run on the Android tablets. There are several aspects of the experiment:

– Letting a medical professional come with a self-chosen workflow, tests the expressibility of the system.
– Letting a new user interact with the workflow creation tool tests the usability of the tool.
– Letting a medical professional use the tool, tests the hypothesis: the domain expert can implement simple workflows, leading to a simpler and more flexible development process, e.g.
  • The domain experts might be able to make simple changes directly without involving the development team.
  • The domain experts can use simple workflows to communicate more directly and efficiently with the development team.

3.1 The Experiment

The experiment, which took a single day, was set-up as follows: DFL had access to a computer where the server, the code generator and example spreadsheets were available. To simplify the interface, all spreadsheets was placed on the desktop and batch commands performed the code generation and server start. To learn the syntax DFL, did a small exercise under instruction by one of the authors a few days before the experiment.

The workflow chosen by DFL model how a healthy woman gets an abortion, according to DFL was ”a simplification of the simplest workflow I could find”.

Id	Name	D	N	S	AN	OPN	Seq	Log	Condition
1.1.1	Nurse evaluation	R	W	R	R	R			
1.1.2	Patient History	W	R	R	R	R			
1.1.3	Extended history	W	R	R	R	R			abnorm
1.1.4	Preoperative treatment	W	R	R	R	R			cyto
1.1.5	Objective	W	R	R	R	R			
1.1.6	Extended objective	W	R	R	R	R			abnorm2
1.1.7	Ultrasound	W	R	R	R	R			
1.1.8	Formalia	W	R	R	R	R			
1.1.9	Extended formalia	W	R	R	R	R			abnorm3
1.2.0	Information for the patient	R	W	R	R	R		1.1.1 - 1.1.9	
1.2.1	Schedule for OP	R	R	W	R	W	1.2.0		

Figure 9. End-user developed workflow (Flow).

The developed workflow is shown in Fig. 9 and Fig. 10. The roles are: Doctor (D), Nurse (N), Secretary (S), Anestesiologist (AN) and operation nurse (OPN).

An example screenshot from the running Android client is shown in Fig. 11.

Id	Input	Action
1.1.1	name height weight bP	

1.1.2	cave ever_birth healthy	if ! healthy then set(abnorm);
		if cave then set(abnorm);
		if ! ever_birth then set(cyto)

1.1.3	cavetx healthtx	
1.1.4	rp_cytotec	

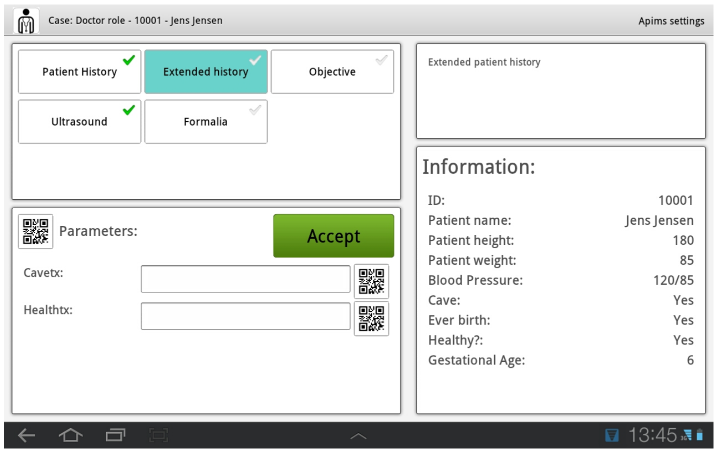
1.1.5	gU_ia stet_c_et_p_ia
	uterus_retroflekteret
		if ! gU_ia then set(abnorm2);
		if ! stet_c_et_p_ia then set(abnorm2)

1.1.6	sttx gutx	
1.1.7	fHR cRL gA	

1.1.8	clamydiatested clamydia_negative
	rhesus_negative signed_form_A
	under_18 gA_under_12
		if ! clamydiatested then set(abnorm3);
		if ! clamydia_negative then set(abnorm3);
		if rhesus_negative then set(abnorm3);
		if under_18 then set(abnorm3);
		if ! signed_form_A then reset(1.1.8);
		if ! gA_under_12 then reset(1.1.8)

1.1.9	rp_antibiotics rp_anti_D signed_form_B	
1.2.0	pt_informeret_samtykke	
1.2.1	op_tid gA_ved_op	

Figure 10. End-user developed workflow (Data).



Cavetx:
Healthtx:

Information:

ID:	10001
Patient name:	Jens Jensen
Patient height:	180
Patient weight:	85
Blood Pressure:	120/85
Cave:	Yes
Ever birth:	Yes
Healthy?:	Yes
Gestational Age:	6

Figure 11. Screenshot from Android client running the experiment workflow.

3.2 Evaluation

Generally the experiment turned out very successfully: DFL was easily able to use the spredsheets to build her own workflow. The instructing author only had to take over one time to fix a problem. Even though the workflow included fairly complex logic, DFL was able to create it without any previous programming experience and despite the unwieldy syntax of the action field. The system seemed expressive enough to create the simple flow, but during the experience DFL asked for more complex logic (e.g. comparison of values) and more presentation control (e.g. grouping of values). The general usability of the tool seemed good, as DFL was able to start developing her workflow almost from the start. Of course there are several points that could be improved (most notably the action field). All in all, it is promising to let a domain expert work directly with the workflow code; maybe not for the full version, but for rapid prototyping.

4 Formal Theory

This section provides the outline of the formal theory and shows the properties which the prototype can ensure. Due to the space limitation, detailed definitions and proofs are left to Appendix and [3]. `ASH: Mention guisync and guivalue?`

Once given global types as a description of global interactions among communicating processes, we can consider the following development steps for validating programs. `NY: Andres can write how each step is related to architecture by citing them?`

Step 1 A programmer describes an intended interaction protocol as global type G with logical predicates, and checks that it is well-formed or not.

Step 2 A programmer generates projections of global type G (called *local types*) onto each participant.

Step 3 She develops program code P, one for the local behaviour of each participant $\mathtt{p}$, incrementally validating its conformance to its local type T by efficient type-checking.

When programs are executed, their interactions are guaranteed to follow the stipulated scenario without deadlocks.

Going back to the running example from § 2, the local type describing the behaviour of each participant can be obtained by projection (Step 2) and following this type, its process is implemented by filling input and output binding of values (Step 3). The local type of the patient and its behaviour described in as a process is given in Fig. 12. There is a clear one-to-one correspondence between type and process: for example, the recursive type μ corresponds to the recursive agent (denoted by def) and the sum type corresponds to the synchronisation (denoted by `sync`). `NY: we need to point the corresponding figures and code in Implementation`

We then type-check processes by following the session typing rules. The typing judgement extends the original one [1] with symmetric sum types. The judgement $\Theta; \Gamma \vdash P \triangleright \Delta$ states that assuming Θ the process P in the environment Γ performs

```
G↾1 = // Local type for Patient
μ  workflow⟨test1:Bool=false,
            test2:Bool=false,
            pre:Bool=false,
            result1:Bool=false,
            result2:Bool=false⟩.
{ Test1 [[not pre]]:
   1?⟨Bool⟩ as x;
   forall y[[x=y]];
    workflow⟨true,test2,pre,x,result2⟩,
  Test2 [[not pre]]:
   1?⟨Bool⟩ as x;
   forall y[[x=y]];
    workflow⟨test1,true,pre,result1,x⟩,
  Prescribe [[test1 and test2 and
             not pre and
             not (result1 and result2)]]:
   3?⟨String⟩ as x;
    workflow⟨test1,test2,true,
             result1,result2⟩,
  Discharge [[test1 and test2 and
     ((result1 and result2) or pre)]]
  end
}
```

```
P_P = // Patient
ā[2..3](p,d,n).
def X⟨t1:Bool,t2:Bool,pre:Bool,
      r1:Bool,r2:Bool⟩
     ((p,d,n): workflow↾1⟨t1,t2,pre,
                          r1,r2⟩)=
  sync((p,d,n),3)
  { Test1 [[not pre]]:
     p?(result);
     X⟨true,t2,pre,result,r2⟩((p,d,n)),
    Test2 [[not pre]]:
     p?(result);
     X⟨t1,true,pre,r1,result⟩((p,d,n)),
   Prescribe [[t1 and t2 and not pre
               and not (r1 and r2)]]:
     p?(prescription);
     X⟨t1,t2,true,r1,r2⟩((p,d,n)),
   Discharge [[t1 and t2 and
             ((r1 and r2) or pre)]]}
    end
  }
    in X⟨false,false,false,false,false⟩((p,d,n))
```

Figure 12. Local type and process for the patient

exactly the session communication described in Δ. By the rules, we can check the running example is typable, i.e. $\Theta; \Gamma \vdash P_P \mid P_D \mid P_N \triangleright \Delta$ where P_D and P_N are the doctor and nurse processes implemented similarly to P_P and $\mid$ denotes the parallel compostion. We end this section by proving *subject reduction theorem*, which guarantees that once process is compiled, then there will be no type error at runtime.

Theorem 1 (Subject reduction). $\mathsf{true}; \Gamma \vdash P \triangleright \emptyset$ *and* $P \to P'$*, then* $\mathsf{true}; \Gamma \vdash P' \triangleright \emptyset$.

The proof can be found in Appendix B and [3]. From this theorem, we can derive many safety properties as corollaries [2, § 5]. The properties which this framework guarantees include: (1) **type safety**: the lack of standard type errors in expressions; (2) **communication safety**: communication error freedom (i.e. a sending action always matched to its corresponding receiving action at the same channel); (3) **session fidelity**: the interactions of a typable process exactly follow the specification described by its global type; and (4) **progress**: once a communication has been established, well-typed programs will never stuck at communication points. The formal definitions and the full proofs of these properties can be found in [3,1,2].

5 Conclusions, Related and Future work

We have successfully merged the symmetric sum types and the type assertions extensions of the multiparty session types into a single type language. This enables the benefits of assertion types – such as value restrictions and more efficient representation of some interaction patterns – in the workflows that can be represented using symmetric sum types. The extended typing judgement still ensure subject reduction. We have implemented the used language and type verification for the assertion language of classical propositional logic.

The theorem provers we have implemented are based on the LK and CFLKF proof systems, but the is a vast abundance of theorem provers available [?] [?] which can enable both more efficient verification (in practice) and more expressive assertion languages. We can even use a resolution [?] based theorem prover or indeed any method that can decide assertion validity, as we do not currently use the derivations for anything. The results we have proved are not as powerful as the ones proved for the original assertions paper [1]. This is because we do not include the assertions in the programs, and therefore assertion violations are not detected during execution which means we cannot prove the *well typed terms do not go wrong* result. This is not related to the extension with symmetric sum types, and thus it should be possible to extend the program syntax with assertions and prove the result.

Acknowledgements This work was funded in part by the Danish Council for Strategic Research, Grant #2106-07-0019, the IT University of Copenhagen and Copenhagen University (the TrustCare project). **We need to thank a physician.**

References

1. Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, LNCS, pages 162–176. Springer, 2011.
2. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
3. Lasse Nielsen. *Regular Expressions and Multiparty Session Types with Applications to Workflow Based Verification of User Interfaces.* PhD thesis, ITU Copenhagen, 2012.
4. Lasse Nielsen, Nobuko Yoshida, and Kohei Honda. Multiparty symmetric sum types. *Arxiv preprint arXiv:1011.6436*, 2010.
5. Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Available from http://thrift.apache.org/.
6. Annette ten Teije, Silvia Miksch, and Peter Lucas. *Computer-based Medical Guidelines and Protocols: A Primer and Currend Trends.* Studies in Health Technology and Informatics. IOS Press, 2008.
7. Wil M.P van der Aalst and Maja Pesic. A declarative approach for flexible business processes management. In *Proceedings DPM 2006*, 2006.
8. Web Services Choreography Working Group. Choreography Description Language. http://www.w3.org/2002/ws/chor/.

A Multiparty Sum Types with Assertions

This appendix lists the omitted definitions and proofs from the main paper.

A.1 Multiparty Session Types

Global Types We start by defining the global types G in Fig. 13. The type $\mathtt{p} \to \mathtt{p}' : k\langle U\rangle\ \mathtt{as}\ x\,\{A\}\,.G'$ expresses that participant $\mathtt{p}$ sends a message of type U along channel k to $\mathtt{p}'$ and then interactions described in G' take place. Here

Global Types

$$\begin{array}{rl} G ::= & \mathtt{p} \to \mathtt{p}' : k\langle S \rangle \;\; \mathtt{as} \;\; x\,\{A\}\,.G' \\ \mid & \mathtt{p} \to \mathtt{p}' : k\langle U \rangle.G' \\ \mid & \mathtt{p} \to \mathtt{p}' : k\{\{A_i\}\, l_i : G_i\}_{i \in I} \\ \mid & \mu t\langle \tilde{x} \rangle(\tilde{e}).G \\ \mid & t\langle \tilde{e} \rangle \\ \mid & \mathsf{end} \\ \mid & \{\{A_l\}\, l : G_l\}_{l \in L;M} \end{array}$$

Local Types

$$\begin{array}{rl} T ::= & k!\langle S \rangle \;\; \mathtt{as} \;\; x\,\{A\}\,; T \\ \mid & k?\langle S \rangle \;\; \mathtt{as} \;\; x\,\{A\}\,; T \\ \mid & k!\langle U \rangle; T \mid k?\langle U \rangle; T \\ \mid & k \oplus \{\{A_l\}\, l : T_l\}_{l \in L} \\ \mid & k \,\&\, \{\{A_l\}\, l : T_l\}_{l \in L} \\ \mid & \mu t\langle \tilde{x} \rangle(\tilde{e}).T \mid t\langle \tilde{e} \rangle \mid \mathsf{end} \\ \mid & \{\{A_l\}\, l : T_l\}_{l \in L;M} \\ \mid & \forall x : S\,\{A\}\,.T \end{array}$$

Message Types

$$U ::= \tilde{S} \mid T@(\mathtt{p}, m, n)$$

Simple Types

$$S ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots \mid \langle G \rangle$$

Figure 13. Global and Local types

$\mathtt{p}, \mathtt{p}', \mathtt{q}, \mathtt{r} \ldots$ denote the *participant*. The $\mathtt{as}\;\; x\,\{A\}$ parts binds occurrences of x in G and A to the value communicated, and states that the value must respect the predicate A. The type $\mathtt{p} \to \mathtt{p}' : k\{\{A_i\}\, l_i : G_i\}_{i \in I}$ expresses that $\mathtt{p}$ sends one of the labels l_i to $\mathtt{p}'$. If l_j is sent, then the predicate A_j must be fulfilled, and the interactions described in G_j take place. Type $\mu t\langle \tilde{x} \rangle.G$ is a recursive type where $\tilde{x}$ is the state, assuming type variables $(t, t', \ldots)$ are guarded in the standard way. We assume that G in the grammar of sorts is closed, i.e., without free type or assertion variables. Type end represents the session termination.

The sum type $\{\{A_l\}\, l : G_l\}_{l \in L;M}$ represents a synchronisation where the labels are taken from the set $L \cup M$. The labels in L are optional, but the labels in M are mandatory and must be accepted by all the participants. If the predicate A_l is false, the label is ignored, and must be rejected. The mandatory labels will be underlined to distinguish them from the optional labels (e.g. $\{l : G_l\}_{l \in \{l1\};\{l2\}} = \{l1 : G_{l1}, \underline{l2} : G_{l2}\}$).

Local Types The local types T are defined in Fig. 13. They describe the communication performed by a single process. Therefore the "from process to process on channel" syntax is simply changed to sending or receiving on a channel. Thus the sending type is $k!\langle U \rangle \;\; \mathtt{as} \;\; x\,\{A\}\,; T$ and represents sending a message of type U on channel k respecting the predicate A, followed by the communication described by T. The type of receiving is $k?\langle U \rangle \;\; \mathtt{as} \;\; x\,\{A\}\,; T$, the type of selecting is $k \oplus \{\{A_l\}\, l : T_l\}_{l \in L}$ and the type of branching is $k \,\&\, \{\{A_l\}\, l : T_l\}_{l \in L}$. The difference from the original assertion paper is that the symmetric sum type constructor $\{\{A_l\}\, l : T_l\}_{l \in L;M}$ is added where L, M satisfies the conditions similar to those of a global sum type, and we have introduced a $\forall x\,\{A\}\,.T$ constructor. This is to capture the local type of message parsing for a participant that is neither the sender or the receiver in a more intuitive way than the original assertion paper. In this case the local type should allow only the behaviour that is valid for all possible messages, and therefore the local type is represented by a forall construct.

The message type $T@(\mathtt{p}, m, n)$ is used for delegation. It describes an open session, and includes information about the participant number $\mathtt{p}$, the number of session channels m, and the number of participants n in the session together with a local type T describing the remaining communication.

$$(\mathtt{p}_0 \to \mathtt{p}_1 : k\langle S\rangle \ \mathtt{as}\ \ x\,\{A\}\,.G')\upharpoonright\mathtt{p} = \begin{cases} m!\langle S\rangle \ \ \mathtt{as}\ \ x\,\{A\}\,;(G'\upharpoonright\mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_0 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \\ m?\langle S\rangle \ \ \mathtt{as}\ \ x\,\{A\}\,;(G'\upharpoonright\mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_1 \text{ and } \mathtt{p} \neq \mathtt{p}_0 \\ \forall x : S\,\{A\}\,.G'\upharpoonright\mathtt{p} & \text{if } \mathtt{p} \neq \mathtt{p}_0 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \end{cases}$$

$$(\mathtt{p}_0 \to \mathtt{p}_1 : k\langle U\rangle.G')\upharpoonright\mathtt{p} = \begin{cases} m!\langle U\rangle;(G'\upharpoonright\mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_0 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \\ m?\langle U\rangle;(G'\upharpoonright\mathtt{p}) & \text{if } \mathtt{p} = \mathtt{p}_1 \text{ and } \mathtt{p} \neq \mathtt{p}_0 \\ G'\upharpoonright\mathtt{p} & \text{if } \mathtt{p} \neq \mathtt{p}_0 \text{ and } \mathtt{p} \neq \mathtt{p}_1 \end{cases}$$

$$(\mathtt{p}_0 \to \mathtt{p}_1 : k\{\{A_j\}\,l_j : G_j\}_{j\in J})\upharpoonright\mathtt{p} = \begin{cases} k \oplus \{\{A_j\}\,l_j : (G_j\upharpoonright\mathtt{p})\}_{j\in J} & \text{if } \mathtt{p} = \mathtt{p}_0 \neq \mathtt{p}_1 \\ k\ \&\ \{\{A_j\}\,l_j : (G_j\upharpoonright\mathtt{p})\}_{j\in J} & \text{if } \mathtt{p} = \mathtt{p}_1 \neq \mathtt{p}_0 \\ \forall\_\left\{\bigvee_{j\in J} A_j\right\}.\max_{\leq_{\mathtt{sub}}} & \text{if } \mathtt{p}_1 \neq \mathtt{p}\ \ \neq \mathtt{p}_2 \\ \quad \{T' \mid \forall j \in J.T' \leq_{\mathtt{sub}} (G_j\upharpoonright\mathtt{p})\} & \end{cases}$$

$$(\{\{A_l\}\,l : G_l\}_{l\in L;L'})\upharpoonright\mathtt{p} = \{\{A_l\}\,l : (G_l\upharpoonright\mathtt{p})\}_{l\in L;L'}$$

$$(\mu t\langle\tilde{x}\rangle(\tilde{e}).G)\upharpoonright\mathtt{p} = \mu t\langle\tilde{x}\rangle(\tilde{e}).(G\upharpoonright\mathtt{p})$$

$$(t\langle\tilde{e}\rangle)\upharpoonright\mathtt{p} = t\langle\tilde{e}\rangle$$

$$(\mathsf{end})\upharpoonright\mathtt{p} = \mathsf{end}$$

Figure 14. Projection from global to local types ($\upharpoonright$)

$P ::= \mathtt{sync}_{\tilde{s},n}\{\{A_l\}\,l : P_l\}_{l\in L}$	synch	$\mid s \lhd l; P$	selection
$\mid 0$	inaction	$\mid s \rhd \{l : P_l\}_{l\in L}$	branching
$\mid \overline{a}[\mathtt{2..n}](\tilde{s}).P$	request	$\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional
$\mid a[\mathtt{p}](\tilde{s}).P$	accept	$\mid P \mid Q$	parallel
$\mid s!\langle\tilde{e}\rangle; P$	sending	$\mid (\nu n)P$	restriction
$\mid s?(\tilde{x}); P$	reception	$\mid \mathsf{def}\ D\ \mathsf{in}\ P$	recursion
$\mid s!\langle\langle\tilde{s}\rangle\rangle; P$	delegation	$\mid X\langle\tilde{e}\tilde{s}\rangle$	variable
$\mid s?((\tilde{s})); P$	catch	$\mid s : \tilde{h}$	queue
$D ::= \{X_i\langle\tilde{x}_i\rangle(\tilde{s}_i) = P_i\}_{i\in I}$	declarations	$v ::= a \mid \mathsf{true} \mid \mathsf{false}$	values
$e ::= v \mid x \mid e\ \mathsf{and}\ e' \mid \ldots$	expressions	$h ::= l \mid \tilde{v} \mid \tilde{s}$	messages
$A ::= e$	assertions		

Figure 15. The process language

A.2 Projection

A global type G is *coherent* [2] if and only if the projection $G\upharpoonright\mathtt{p}$ is defined for all participants, and G does not allow racing conditions (linearity). We only consider coherent global types.

A.3 The process language

This subsection introduces the syntax (Fig. 15) of the asynchronous multiparty session π-calculus [2] with the new $\mathtt{sync}$ primitive, and the judgement $P \to P'$ (Fig. 16, where $e \downarrow v$ denotes the evaluation of the expression e to the value v) describing the small-step semantics for processes. The syntax defines the values: $\{v, w, \ldots\}$, expressions: $\{e, e', \ldots\}$, assertions $\{A, B, \ldots\}$ and processes: $\{P, Q, \ldots\}$ from the sets of channel names: $\{a, b, \ldots\}$, value variables: $\{x, y, \ldots\}$, session channels: $\{s, t, \ldots\}$, labels: $\{l, m, \ldots\}$ and process variables: $\{X, Y, \ldots\}$.

Session request ($\overline{a}[\mathtt{2..n}](\tilde{s}).P$) initiates a session with channels $\tilde{s}$ (where $\tilde{s}$ denotes a vector $s_1 \ldots s_n$) over the public channel a with the other $n - 1$ partic-

$$\frac{[\textsc{Link}]}{\overline{a}[2..n](\tilde{s}).P_1|a[2](\tilde{s}).P_2|\ldots|a[n](\tilde{s}).P_n \to (\nu\tilde{s})(P_1|P_2|\ldots|P_n|s_1:\emptyset|\ldots|s_m:\emptyset)}$$

$$\frac{[\textsc{Send}] \quad \tilde{e}\downarrow\tilde{v}}{s!\langle\tilde{e}\rangle;P|s:\tilde{h} \to P|s:\tilde{h}\cdot\tilde{v}} \qquad \frac{[\textsc{Recv}]}{s?(\tilde{x});P|s:\tilde{v}\cdot\tilde{h} \to P[\tilde{v}/\tilde{x}]|s:\tilde{h}}$$

$$\frac{[\textsc{Label}]}{s\triangleleft l;P|s:\tilde{h} \to P|s:\tilde{h}\cdot l} \qquad \frac{[\textsc{Branch}] \quad j\in I}{s\triangleright\{l_i:P_i\}_{i\in I}|s:l_j\cdot\tilde{h} \to P_j|s:\tilde{h}}$$

$$\frac{[\textsc{Deleg}]}{s!\langle\langle\tilde{t}\rangle\rangle;P|s:\tilde{h} \to P|s:\tilde{h}\cdot\tilde{t}} \qquad \frac{[\textsc{SRec}]}{s?((\tilde{t}));P|s:\tilde{t}\cdot\tilde{h} \to P|s:\tilde{h}}$$

$$\frac{[\textsc{IfT}] \quad e\downarrow\mathsf{true}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to P} \qquad \frac{[\textsc{IfF}] \quad e\downarrow\mathsf{false}}{\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to Q}$$

$$\frac{[\textsc{Def}] \quad \tilde{e}\downarrow\tilde{v} \quad X\langle\tilde{x}\tilde{s}\rangle = P\in D}{\mathsf{def}\ D\ \mathsf{in}\ X\langle\tilde{e}\tilde{s}\rangle|Q \to \mathsf{def}\ D\ \mathsf{in}\ P[\tilde{v}/\tilde{x}]|Q} \qquad \frac{[\textsc{Scop}] \quad P\to P'}{(\nu n)P \to (\nu n)P'}$$

$$\frac{[\textsc{Par}] \quad P\to P'}{P|Q \to P'|Q} \qquad \frac{[\textsc{Defin}] \quad P\to P'}{\mathsf{def}\ D\ \mathsf{in}\ P \to \mathsf{def}\ D\ \mathsf{in}\ P'} \qquad \frac{[\textsc{Str}] \quad P'\equiv \quad \to Q\equiv Q'}{P'\to Q'}$$

$$\frac{[\textsc{Sync}] \quad h\in\bigcap_{i=1}^{n} L_i \quad A_{1h}\downarrow\mathsf{true} \quad \cdots \quad A_{nh}\downarrow\mathsf{true}}{\mathtt{sync}_{\tilde{s},n}\{\{A_{1l}\}\,l:P_{1l}\}_{l\in L_1} \mid \ldots \mid \mathtt{sync}_{\tilde{s},n}\{\{A_{nl}\}\,l:P_{nl}\}_{l\in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}$$

Figure 16. The reduction rules

ipants of shape $a[\mathtt{p}](\tilde{s}).Q_{\mathtt{p}}$ for $\mathtt{p}$ from 2 to n. ([Link] in Fig. 16). Asynchronous communication in an established session is performed by sending and receiving values ([Send,Recv]), transferring a session using session delegation and reception ([Deleg,SRec]), and label selection and branching ([Label,Branch]), where the branching process offers a number of labels and the selecting process chooses one of them.

The new $\mathtt{sync}_{\tilde{s},n}\{\{A_l\}l:P_l\}_{l\in L}$ constructor is interpreted as the process participating in a plenum decision between all the n processes in the session $\tilde{s}$ reaching a common decision h from L, which all the processes accept since the assertions evaluate to true. Afterwards each process $\mathtt{p}$ proceeds as described in $P_{\mathtt{p}h}$. In [Sync] in Fig. 16, h in the premise denotes the common label. In [Sync], the processes cannot perform the synchronisation if they do not accept some common label, in which case the processes will be stuck. We also need to know how many participants are in the session in order to know when the synchronisation can step; otherwise the processes will be stuck, or some processes will be left behind. The typing system introduced in the next section ensures that sync satisfies such conditions.

Workflow example (2): Processes We give implementations of the patient, doctor and nurse in the workflow from § 2 in Fig. 17.

A.4 Typing Processes

The typing judgement extends the original one [1] with symmetric sum types. The judgement $\Theta;\Gamma \vdash P \triangleright \Delta$ states that assuming Θ the process P in the environment Γ performs exactly the session communication described in Δ. Formally, the environments are defined as:

$$\Gamma ::= \emptyset \mid \Gamma, u:\langle G\rangle \mid \Gamma, X:\tilde{S}\tilde{T} \qquad \Delta ::= \emptyset \mid \Delta, \tilde{s}:T@(\mathtt{p},n)$$

```
P_P = // Patient
ā[2..3](p,d,n).
def X⟨t1:Bool,t2:Bool,pre:Bool,
        r1:Bool,r2:Bool⟩
       ((p,d,n): workflow↾1⟨t1,t2,pre,
                              r1,r2⟩)=
  sync((p,d,n),3)
  { Test1 [[not pre]]:
      p?(result);
      X⟨true,t2,pre,result,r2⟩((p,d,n)),
    Test2 [[not pre]]:
      p?(result);
      X⟨t1,true,pre,r1,result⟩((p,d,n)),
    Prescribe [[t1 and t2 and not pre
                and not (r1 and r2)]]:
      p?(prescription);
      X⟨t1,t2,true,r1,r2⟩((p,d,n)),
    Discharge [[t1 and t2 and
          ((r1 and r2) or pre)]]}
     end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

```
P_D = // Doctor
a[2](p,d,n).
def X⟨t1:Bool,t2:Bool,pre:Bool,
        r1:Bool,r2:Bool⟩
       ((p,d,n): workflow↾2⟨t1,t2,pre,
                              r1,r2⟩)=
  sync((p,d,n),3)
  { Test1 [[not pre]]:
      d?(result);
      X⟨true,t2,pre,result,r2⟩((p,d,n)),
    Test2 [[not pre]]:
      d?(result);
      X⟨t1,true,pre,r1,result⟩((p,d,n)),
    Prescribe [[t1 and t2 and
        not pre and not (r1 and r2)]]:
      p!⟨eResult⟩; n!⟨eResult⟩;
      X⟨t1,t2,true,r1,r2⟩((p,d,n)),
    Discharge [[t1 and t2 and
              ((r1 and r2) or pre)]]}
     end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

```
P_N = // Nurse
a[3](p,d,n).
def X⟨t1:Bool,t2:Bool,adm:Bool,r1:Bool,r2:Bool⟩
       ((p,d,n): workflow↾3⟨t1,t2,adm,r1,r2⟩)=
  sync((p,d,n),3)
  { Test1 [[]]:
      p!⟨eResult⟩; d!⟨eResult⟩; X⟨true,t2,adm,eResult,r2⟩((p,d,n)),
    Test2 [[]]:
      p!⟨eResult⟩; d!⟨eResult⟩; X⟨t1,true,adm,r1,eResult⟩((p,d,n)),
    Prescribe [[t1 and t2 and not adm and not (r1 and r2):
      X⟨t1,t2,true,r1,r2⟩((p,d,n)),
    Discharge [[t1 and t2 and ((r1 and r2) or adm)]]:
     end
  }
in X⟨false,false,false,false,false⟩((p,d,n))
```

Figure 17. Implementation of the participants in the workflow from Fig. 1 and Fig. 4

The environment Γ contains the global types for shared channels u, and process variables X and the session type environment Δ contains the remaining session communication in Fig. 13, where $\tilde{s} : T@(p, n)$ means $\tilde{s}$ is an open session with n participants, where T describes the remaining communication for participant $\mathtt{p}$. Since the process is reduced by each rule-application, the typability question $\Theta; \Gamma \vdash P \triangleright \Delta$ is decidable when $\vdash A$ is decidable. We shall leave all typing and logical rules in Appendix A.5.

Using the global type and its projections we can now typecheck the processes.

A.5 Typing Rules

The main rules are included in Fig. 18. The local types now carry information about the number of participants n and channels m. The number of participants and channels is determined at the session initialisation in the rules [Mcast] and [Macc], where $\mathsf{sid}(G)$ denotes channels that appear in G and $\mathsf{pid}(G)$ denotes the participants that appear in G. The rule [Sync] checks that the synchronisation uses the correct number of participants, the accepted branches includes the mandatory ones exactly (if and only if) when their predicate is fulfilled, and does not accept (only if) the optional ones with a false predicate. It is checked that there is always an active mandatory option, and finally that each accepted branch is typed with the correct communication.

A.6 Process congruence

The process semantics uses the notion of process equivalence ($\equiv$), and this is included in Figure 19.

A.7 Symmetric sum types

We include the full set of typing rules. The typing system uses the notion of type projection ($\upharpoonright$) and the notion of subtyping ($\leq$) of session-environments which consists of subtyping of each of the used local types defined in [3].

The expression typing rules ($\Gamma \vdash e : S$)

$$\frac{[\textsc{True}]}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{[\textsc{False}]}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{[\textsc{Var}]}{\Gamma, x : S \vdash x : S} \qquad \frac{[\textsc{Not}]\ \Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \mathsf{not}\ e : \mathsf{bool}}$$

$$\frac{[\textsc{And}]\ \Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1\ \mathsf{and}\ e_2 : \mathsf{bool}} \qquad \frac{[\textsc{Or}]\ \Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1\ \mathsf{or}\ e_2 : \mathsf{bool}} \qquad \cdots$$

The typing rules ($\Theta; \Gamma \vdash P \triangleright \Delta$)

$$\frac{[\textsc{Subs}]\ \Theta; \Gamma \vdash P \triangleright \Delta \quad \vdash \Theta \Rightarrow \Delta \leq \Delta'}{\Theta; \Gamma \vdash P \triangleright \Delta'}$$

$$\frac{\begin{array}{c}\forall l \in L'' : \Theta \wedge A_l; \Gamma \vdash P_l \triangleright \Delta, \tilde{s} : T_l@(\mathtt{p}, n) \quad L'' \subseteq L \cup L' \\ \forall l \in L \backslash L'' :\vdash \Theta \Rightarrow \neg B_l \quad \forall l \in L'' :\vdash \Theta \Rightarrow (A_l \Rightarrow B_l) \\ [\textsc{Sync}] \qquad \forall l \in L :\vdash \Theta \Rightarrow (B_l \Rightarrow A_l) \quad \vdash \Theta \Rightarrow \bigvee_{l \in L} B_l\end{array}}{\Theta; \Gamma \vdash \mathtt{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l \in L''} \triangleright \Delta, \tilde{s} : \{\{B_l\}\, l : T_l\}_{l \in L; L'}@(\mathtt{p}, n)}$$

Figure 18. Selected typing rules

$$\textsc{[Sync]}\ \frac{\begin{array}{c}\forall l \in L'' : \Theta \wedge A_l; \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(\mathtt{p},n) \quad L'' \subseteq L \cup L' \\ \forall l \in L\setminus L'' :\vdash \Theta \Rightarrow \neg B_l \quad \forall l \in L'' :\vdash \Theta \Rightarrow (A_l \Rightarrow B_l) \\ \forall l \in L :\vdash \Theta \Rightarrow (B_l \Rightarrow A_l) \quad \vdash \Theta \Rightarrow \bigvee_{l\in L} B_l\end{array}}{\Theta;\Gamma \vdash \mathtt{sync}_{\tilde{s},n}\{\{A_l\}\, l : P_l\}_{l\in L''} \rhd \Delta, \tilde{s} : \{\{B_l\}\, l : T_l\}_{l\in L;L'}@(\mathtt{p},n)}$$

$$\textsc{[Mcast]}\ \frac{\begin{array}{ll}\Gamma \vdash a : \langle G\rangle & |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ \Theta;\Gamma \vdash P \rhd \Delta, \tilde{s} : (G\restriction 1)@(\mathtt{1},n) & n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash \overline{a}[\mathtt{2..n}](\tilde{s}).P \rhd \Delta}\ (fv(G\restriction 1) = \emptyset)$$

$$\textsc{[Macc]}\ \frac{\begin{array}{ll}\Gamma \vdash a : \langle G\rangle & |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ \Theta;\Gamma \vdash P \rhd \Delta, \tilde{s} : (G\restriction \mathtt{p})@(\mathtt{p},n) & n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash a[\mathtt{p}](\tilde{s}).P \rhd \Delta}\ (fv(G\restriction \mathtt{P}) = \emptyset)$$

$$\textsc{[SendA]}\ \frac{\Gamma \vdash e : S \quad \Theta;\Gamma \vdash P \rhd \Delta, s : T[e/x]@(\mathtt{p},n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta;\Gamma \vdash s_k!\langle e\rangle; P \rhd \Delta, \tilde{s} : k!\langle S\rangle\ \mathtt{as}\ x\,\{A\}\,; T@(\mathtt{p},n)}$$

$$\textsc{[RcvA]}\ \frac{\Theta \wedge A; \Gamma, x : S \vdash P \rhd \Delta, \tilde{s} : T@(\mathtt{p},n)}{\Theta;\Gamma \vdash s_k?(x); P \rhd \Delta, \tilde{s} : k?\langle S\rangle\ \mathtt{as}\ x\,\{A\}\,; T@(\mathtt{p},n)}\ (x \notin fv(\Theta) \cup fv(\Delta))$$

$$\textsc{[Sel]}\ \frac{\Theta;\Gamma \vdash P \rhd \Delta, \tilde{s} : T@(\mathtt{p},n) \quad h \in L \quad \vdash \Theta \Rightarrow A_h}{\Theta;\Gamma \vdash s_k \lhd h; P \rhd \Delta, \tilde{s} : k \oplus \{\{A_l\}\, l : T_l\}_{l\in L}@(\mathtt{p},n)}$$

$$\textsc{[Branch]}\ \frac{\forall l \in L : \Theta \wedge A_l; \Gamma \vdash P_l \rhd \Delta, \tilde{s} : T_l@(\mathtt{p},n)}{\Theta;\Gamma \vdash s_k \rhd \{l : P_l\}_{l\in L} \rhd \Delta, \tilde{s} : k\&\{\{A_l\}\, l : T_l\}_{l\in L}@(\mathtt{p},n)}$$

$$\textsc{[Conc]}\ \frac{\Theta;\Gamma \vdash P \rhd \Delta \quad \Theta;\Gamma \vdash Q \rhd \Delta'}{\Theta;\Gamma \vdash P|Q \rhd \Delta \circ \Delta'}\ (\mathsf{dom}(\Delta) \cap \mathsf{dom}(\Delta') = \emptyset)$$

$$\textsc{[Subs]}\ \frac{\Theta;\Gamma \vdash P \rhd \Delta \quad \vdash \Theta \Rightarrow \Delta \le \Delta'}{\Theta;\Gamma \vdash P \rhd \Delta'}$$

Figure 19. Process congruence ($\equiv$)

The relation $\equiv$ is defined as the smallest *congruence* relation satisfying

$P|0 \equiv P$ $\qquad$ $P|Q \equiv Q|P$

$P|(Q|R) \equiv (P|Q)|R$ $\qquad$ $(\nu n)P|Q \equiv (\nu n)(P|Q)$ if $n \notin \mathsf{fn}(Q)$

$(\nu n n')P \equiv (\nu n' n)P$ $\qquad$ $(\nu n)0 \equiv 0$

def D in $0 \equiv 0$ $\qquad$ $(\nu s_1..s_n)\Pi_i s_i : \emptyset \equiv 0$

def D in $(\nu n)P \equiv (\nu n)$def D in P if $n \notin \mathsf{fn}(D)$

(def D in P) $|\ Q \equiv$ def D in $(P \mid Q)$ if $\mathsf{dpv}(D) \cap \mathsf{fpv}(Q) = \emptyset$

def D in def D' in $P \equiv$ def D and D' in P if $\mathsf{dpv}(D) \cap \mathsf{dpv}(D') = \emptyset$

$$\textsc{[Mcast]}\ \frac{\Gamma \vdash a : \langle G\rangle \quad \Theta;\Gamma \vdash P \rhd \Delta, \tilde{s} : (G\restriction 1)@(\mathtt{1},n) \quad \begin{array}{r}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash \overline{a}[\mathtt{2..n}](\tilde{s}).P \rhd \Delta}$$

$$\textsc{[Macc]}\ \frac{\Gamma \vdash a : \langle G\rangle \quad \Theta;\Gamma \vdash P \rhd \Delta, \tilde{s} : (G\restriction \mathtt{p})@(\mathtt{p},n) \quad \begin{array}{r}|\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ n = \mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma \vdash a[\mathtt{p}](\tilde{s}).P \rhd \Delta}$$

$$\textsc{[SendA]}\ \frac{\Gamma \vdash e : S \quad \Theta;\Gamma \vdash P \rhd \Delta, s : T[e/x]@(\mathtt{p},n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta;\Gamma \vdash s_k!\langle e\rangle; P \rhd \Delta, \tilde{s} : k!\langle S\rangle\ \mathtt{as}\ x\,\{A\}\,; T@(\mathtt{p},n)}$$

$$\frac{[\textsc{RcvA}]\qquad \Theta\wedge A;\Gamma,x:S\vdash P\rhd\Delta,\tilde{s}:T@(\mathtt{p},n)}{\Theta;\Gamma\vdash s_k?(x);P\rhd\Delta,\tilde{s}:k?\langle S\rangle\ \ \mathtt{as}\ \ x\,\{A\}\,;T@(\mathtt{p},n)}\,(x\notin fv(\Theta)\cup fv(\Delta))$$

$$\frac{[\textsc{Send}]\ \forall j.\Gamma\vdash e_j:S_j\quad \Theta;\Gamma\vdash P\rhd\Delta,s:T@(\mathtt{p},n)}{\Theta;\Gamma\vdash s_k!\langle\tilde{e}\rangle;P\rhd\Delta,\tilde{s}:k!\langle\tilde{S}\rangle;T@(\mathtt{p},n)}$$

$$\frac{[\textsc{Rcv}]\quad \Theta;\Gamma,\tilde{x}:\tilde{S}\vdash P\rhd\Delta,\tilde{s}:T@(\mathtt{p},n)}{\Theta;\Gamma\vdash s_k?(\tilde{x});P\rhd\Delta,\tilde{s}:k?\langle\tilde{S}\rangle;T@(\mathtt{p},n)}\,(\tilde{x}\cap(fv(\Theta)\cup fv(\Delta))=\emptyset)$$

$$\frac{[\textsc{If}]\ \Gamma\vdash e:\mathsf{bool}\quad \Theta\wedge e;\Gamma\vdash P\rhd\Delta\quad \Theta\wedge\neg e;\Gamma\vdash Q\rhd\Delta}{\Theta;\Gamma\vdash\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q\rhd\Delta}$$

$$\frac{[\textsc{Deleg}]\qquad\qquad \Theta;\Gamma\vdash P\rhd\Delta,\tilde{s}:T@(\mathtt{p},n)}{\Theta;\Gamma\vdash s_k!\langle\!\langle\tilde{t}\rangle\!\rangle;P\rhd\Delta,\tilde{s}:k!\langle T'@(\mathtt{p'},|\tilde{t}|,n')\rangle;T@(\mathtt{p},n),\tilde{t}:T'@(\mathtt{p'},n')}$$

$$\frac{[\textsc{Srec}]\qquad \Theta;\Gamma\vdash P\rhd\Delta,\tilde{s}:T@(\mathtt{p},n),\tilde{t}:T'@(\mathtt{p'},n')}{\Theta;\Gamma\vdash s_k?((\tilde{t}));P\rhd\Delta,\tilde{s}:k?\langle T'@(\mathtt{p'},|\tilde{t}|,n')\rangle;T@(\mathtt{p},n)}$$

$$\frac{[\textsc{Sel}]\ \Theta;\Gamma\vdash P\rhd\Delta,\tilde{s}:T@(\mathtt{p},n)\quad h\in L\quad \vdash\Theta\Rightarrow A_h}{\Theta;\Gamma\vdash s_k\lhd h;P\rhd\Delta,\tilde{s}:k\oplus\{\{A_l\}\,l:T_l\}_{l\in L}@(\mathtt{p},n)}$$

$$\frac{[\textsc{Branch}]\quad \forall l\in L:\Theta\wedge A_l;\Gamma\vdash P_l\rhd\Delta,\tilde{s}:T_l@(\mathtt{p},n)}{\Theta;\Gamma\vdash s_k\rhd\{l:P_l\}_{l\in L}\rhd\Delta,\tilde{s}:k\&\{\{A_l\}\,l:T_l\}_{l\in L}@(\mathtt{p},n)}$$

$$\frac{[\textsc{Conc}]\ \Theta;\Gamma\vdash P\rhd\Delta\quad \Theta;\Gamma\vdash Q\rhd\Delta'}{\Theta;\Gamma\vdash P|Q\rhd\Delta\circ\Delta'}\,(\mathsf{dom}(\Delta)\cap\mathsf{dom}(\Delta')=\emptyset)$$

$$\frac{[\textsc{Inact}]\ \Delta\quad \mathsf{end\ only}}{\Theta;\Gamma\vdash 0\rhd\Delta}\qquad\qquad \frac{[\textsc{Nres}]\ \Theta;\Gamma,a:\langle G\rangle\vdash P\rhd\Delta}{\Theta;\Gamma\vdash(\nu a)P\rhd\Delta}$$

$$\frac{[\textsc{Var}]\qquad\qquad \forall j.\Gamma\vdash e_j:S_j\quad \Delta\quad \mathsf{end\ only}}{\Theta;\Gamma,X:(\tilde{x}:\tilde{S})T@\widetilde{(\mathtt{p},n)}\vdash X\langle\tilde{e}\rangle(\tilde{s}_1...\tilde{s}_{|\tilde{T}|})\rhd\Delta,\tilde{T}[\tilde{e}/\tilde{x}]@(\mathtt{p},n)}$$

$$\frac{[\textsc{Def}]\ \Theta;\Gamma,X:(\tilde{x}:\tilde{S})T@\widetilde{(\mathtt{p},n)},\tilde{x}:\tilde{S}\vdash P\rhd T@\widetilde{(\mathtt{p},n)}\quad \Theta;\Gamma,X:(\tilde{x}:\tilde{S})T@\widetilde{(\mathtt{p},n)}\vdash Q\rhd\Delta}{\Theta;\Gamma\vdash\mathsf{def}\ X\langle\tilde{x}\rangle(\tilde{s}_1,...,\tilde{s}_{|\tilde{T}|})=P\ \mathsf{in}\ Q\rhd\Delta}$$

The runtime typing rules ($\Theta;\Gamma\vdash P\rhd_{\tilde{t}}\Delta$) Where Δ in the static typing rules represents a map from $\tilde{s}$ to $T@(\mathtt{p},n)$, the runtime typing rules uses Δ as a map from $(\tilde{s},\mathtt{p})$ to $T@(\mathtt{p},n)$.

The extra $\tilde{t}$ is used to ensure that there is exactly one queue for each session channel in each open session.

$$\frac{[\textsc{Subs}]\ \Theta;\Gamma\vdash P\rhd_{\tilde{t}}\Delta\quad \vdash\Theta\Rightarrow\Delta\leq\Delta'}{\Theta;\Gamma\vdash P\rhd_{\tilde{t}}\Delta'}$$

$$\frac{\begin{array}{c}\forall l\in L'':\Theta\wedge A_l;\Gamma\vdash P_l\rhd_{\tilde{t}}\Delta,\tilde{s}:T_l@(\mathtt{p},n)\quad L''\subseteq L\cup L'\\ \forall l\in L\backslash L'':\vdash\Theta\Rightarrow\neg B_l\quad \forall l\in L'':\vdash\Theta\Rightarrow(A_l\Rightarrow B_l)\\ [\textsc{Sync}]\qquad\quad \forall l\in L:\vdash\Theta\Rightarrow(B_l\Rightarrow A_l)\quad \vdash\Theta\Rightarrow\bigvee_{l\in L}B_l\end{array}}{\Theta;\Gamma\vdash\mathtt{sync}_{\tilde{s},n}\{\{A_l\}\,l:P_l\}_{l\in L''}\rhd_{\tilde{t}}\Delta,\tilde{s}:\{\{B_l\}\,l:T_l\}_{l\in L;L'}@(\mathtt{p},n)}$$

$$\frac{[\textsc{Mcast}]\ \Gamma\vdash a:\langle G\rangle\quad \Theta;\Gamma\vdash P\rhd_{\tilde{t}}\Delta,\tilde{s}:(G\upharpoonright 1)@(\mathtt{1},n)\quad \begin{array}{r}|\tilde{s}|=\mathsf{max}(\mathsf{sid}(G))\\ n=\mathsf{max}(\mathsf{pid}(G))\end{array}}{\Theta;\Gamma\vdash\overline{a}[\mathtt{2..n}](\tilde{s}).P\rhd_{\tilde{t}}\Delta}$$

$$\frac{[\textsc{Macc}]\ \Gamma \vdash a : \langle G \rangle \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : (G \upharpoonright \mathtt{p})@(\mathtt{p}, n) \quad \begin{array}{c} |\tilde{s}| = \mathsf{max}(\mathsf{sid}(G)) \\ n = \mathsf{max}(\mathsf{pid}(G)) \end{array}}{\Theta; \Gamma \vdash a[\mathtt{p}](\tilde{s}).P \rhd_{\tilde{t}} \Delta}$$

$$\frac{[\textsc{SendA}]\ \Gamma \vdash e : S \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, s : T[e/x]@(\mathtt{p}, n) \quad \vdash \Theta \Rightarrow A[e/x]}{\Theta; \Gamma \vdash s_k!\langle e \rangle; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k!\langle S \rangle \ \mathtt{as}\ \ x\,\{A\}\,; T@(\mathtt{p}, n)}$$

$$\frac{[\textsc{RcvA}] \qquad \Theta \wedge A; \Gamma, x : S \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(\mathtt{p}, n)}{\Theta; \Gamma \vdash s_k?(x); P \rhd_{\tilde{t}} \Delta, \tilde{s} : k?\langle S \rangle \ \mathtt{as}\ \ x\,\{A\}\,; T@(\mathtt{p}, n)}\,(x \notin fv(\Theta) \cup fv(\Delta))$$

$$\frac{[\textsc{Send}]\ \forall j. \Gamma \vdash e_j : S_j \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, s : T@(\mathtt{p}, n)}{\Theta; \Gamma \vdash s_k!\langle \tilde{e} \rangle; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k!\langle \tilde{S} \rangle; T@(\mathtt{p}, n)}$$

$$\frac{[\textsc{Rcv}] \quad \Theta; \Gamma, \tilde{x} : \tilde{S} \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(\mathtt{p}, n)}{\Theta; \Gamma \vdash s_k?(\tilde{x}); P \rhd_{\tilde{t}} \Delta, \tilde{s} : k?\langle \tilde{S} \rangle; T@(\mathtt{p}, n)}\,(\tilde{x} \cap (fv(\Theta) \cup fv(\Delta)) = \emptyset)$$

$$\frac{[\textsc{If}]\ \Gamma \vdash e : \mathsf{bool} \quad \Theta \wedge e; \Gamma \vdash P \rhd_{\tilde{t}} \Delta \quad \Theta \wedge \neg e; \Gamma \vdash Q \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \rhd_{\tilde{t}} \Delta}$$

$$\frac{[\textsc{Deleg}] \qquad\qquad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(\mathtt{p}, n)}{\Theta; \Gamma \vdash s_k!\langle\langle \tilde{t} \rangle\rangle; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k!\langle T'@(\mathtt{p'}, |\tilde{t}|, n') \rangle; T@(\mathtt{p}, n), \tilde{t} : T'@(\mathtt{p'}, n')}$$

$$\frac{[\textsc{Srec}] \quad \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(\mathtt{p}, n), \tilde{t} : T'@(\mathtt{p'}, n')}{\Theta; \Gamma \vdash s_k?((\tilde{t})); P \rhd_{\tilde{t}} \Delta, \tilde{s} : k?\langle T'@(\mathtt{p'}, |\tilde{t}|, n') \rangle; T@(\mathtt{p}, n)}$$

$$\frac{[\textsc{Sel}]\ \Theta; \Gamma \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : T@(\mathtt{p}, n) \quad h \in L \quad \vdash \Theta \Rightarrow A_h}{\Theta; \Gamma \vdash s_k \lhd h; P \rhd_{\tilde{t}} \Delta, \tilde{s} : k \oplus \{\{A_l\}\, l : T_l\}_{l \in L}@(\mathtt{p}, n)}$$

$$\frac{[\textsc{Branch}] \quad \forall l \in L : \Theta \wedge A_l; \Gamma \vdash P_l \rhd_{\tilde{t}} \Delta, \tilde{s} : T_l@(\mathtt{p}, n)}{\Theta; \Gamma \vdash s_k \rhd \{l : P_l\}_{l \in L} \rhd_{\tilde{t}} \Delta, \tilde{s} : k \& \{\{A_l\}\, l : T_l\}_{l \in L}@(\mathtt{p}, n)}$$

$$\frac{[\textsc{Conc}]\ \Theta; \Gamma \vdash P \rhd_{\tilde{t}_1} \Delta \quad \Theta; \Gamma \vdash Q \rhd_{\tilde{t}_1} \Delta' \quad \Delta \asymp \Delta' \quad \tilde{t}_1 \cap \tilde{t}_2 = \emptyset}{\Theta; \Gamma \vdash P | Q \rhd_{\tilde{t}_1 \cup \tilde{t}_2} \Delta \circ \Delta'}$$

$$\frac{[\textsc{Inact}]\ \Delta \quad \mathsf{end\ only}}{\Theta; \Gamma \vdash 0 \rhd_{\tilde{t}} \Delta} \qquad\qquad \frac{[\textsc{Nres}]\ \Theta; \Gamma, a : \langle G \rangle \vdash P \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash (\nu a)P \rhd_{\tilde{t}} \Delta}$$

$$\frac{[\textsc{Cres}]\ \Theta; \Gamma, \vdash P \rhd_{\tilde{t}} \Delta, \tilde{s} : \{T_\mathtt{p}@(\mathtt{p}, n)\}_{\mathtt{p}=1}^{n} \quad \{T_\mathtt{p}@(\mathtt{p}, n)\}_{\mathtt{p}=1}^{n}\ \text{coherent} \quad \tilde{s} \subseteq \tilde{t}}{\Theta; \Gamma \vdash (\nu \tilde{s})P \rhd_{\tilde{t} \setminus \tilde{s}} \Delta}$$

$$\frac{[\textsc{Var}] \qquad\qquad \forall j. \Gamma \vdash e_j : S_j \quad \Delta \quad \mathsf{end}\ \text{only}}{\Theta; \Gamma, X : (\tilde{x} : \tilde{S})T@(\widetilde{\mathtt{p}, n}) \vdash X\langle \tilde{e} \rangle(\tilde{s}_1 ... \tilde{s}_{|\tilde{T}|}) \rhd_{\tilde{t}} \Delta, \tilde{T}[\tilde{e}/\tilde{x}]@(\mathtt{p}, n)}$$

$$\frac{[\textsc{Def}]\ \Theta; \Gamma, X : (\tilde{x} : \tilde{S})T@(\widetilde{\mathtt{p}, n}), \tilde{x} : \tilde{S} \vdash P \rhd_{\emptyset} T@(\widetilde{\mathtt{p}, n}) \quad \Theta; \Gamma, X : (\tilde{x} : \tilde{S})T@(\widetilde{\mathtt{p}, n}) \vdash Q \rhd_{\tilde{t}} \Delta}{\Theta; \Gamma \vdash \mathsf{def}\ X\langle \tilde{x} \rangle(\tilde{s}_1, ..., \tilde{s}_{|\tilde{T}|}) = P\ \mathsf{in}\ Q \rhd_{\tilde{t}} \Delta}$$

Plus queue rules

B Proofs for subject reduction theorem

We include the proof of subject reduction in Proof B2. The proof uses Lemma B1 which is an extension of Lemma 5.18 from [2]. It states that the [Subs] rules can be propagated upwards to the [Sel] and [Branch] rules.

Lemma B1 (Extension of Lemma 5.18 from [2]: Permutation)**.**

(1) If $\dfrac{[\textsc{Subs}]\ \dfrac{[\textsc{Subs}]\ \dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde t} \Delta}}{\Gamma \vdash P \rhd_{\tilde t} \Delta'}}{\Gamma \vdash P \rhd_{\tilde t} \Delta''}$ *then* $\dfrac{[\textsc{Subs}]\ \dfrac{\mathcal{D}}{\Gamma \vdash P \rhd_{\tilde t} \Delta}}{\Gamma \vdash P \rhd_{\tilde t} \Delta''}$.

(2) If $\dfrac{[\textsc{Subs}]\ \dfrac{[\mathrm{X}]\quad \mathcal{D}}{\Gamma \vdash P \rhd_{\tilde t} \Delta}}{\Gamma \vdash P \rhd_{\tilde t} \Delta'}$ *and the second last rule-application X is not Sel or Branch then the last two rule-applications can be permuted.*

Proof:

(1) Is immediate because $\leq_{\mathrm{sub}}$ is transitive.

(2) Is proved for each possible rule X. This is done as in the original proof. There is one new case, and we will prove it now.

Sync: In this case we consider a derivation

$$\dfrac{[\textsc{Subs}]\ \dfrac{[\textsc{Sync}]\quad \forall l \in L''\quad \dfrac{\mathcal{D}_l}{\Theta \wedge A_l; \Gamma \vdash P_l \rhd_{\tilde t} \Delta, \tilde s : \{T_l @ (\mathtt{p}, n)\}}\quad \cdots}{\Theta; \Gamma \vdash \mathtt{sync}_{\tilde s, n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde t} \Delta, \tilde s : \{\{\{A_l\}\, l : T_l\}_{l \in L; L'} @ (\mathtt{p}, n)\}}}{\Theta; \Gamma \vdash \mathtt{sync}_{\tilde s, n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde t} \Delta', \tilde s : \{\{\{A_l\}\, l : T'_l\}_{l \in L; L'} @ (\mathtt{p}, n)\}}$$

where $T_l \leq_{\mathrm{sub}} T'_l$ for each $l \in L''$ and $\Delta \leq_{\mathrm{sub}} \Delta'$. We can therefore create

$$\dfrac{[\textsc{Sync}]\quad \forall l \in L''\quad \dfrac{[\textsc{Subs}]\ \dfrac{\mathcal{D}_l}{\Gamma \vdash P_l \rhd_{\tilde t} \Delta, \tilde s : \{T_l @ (\mathtt{p}, n)\}}}{\Theta \wedge A_l; \Gamma \vdash l : P_l \rhd_{\tilde t} \Delta', \tilde s : \{T'_l @ (\mathtt{p}, n)\}}\quad \cdots}{\Theta; \Gamma \vdash \mathtt{sync}_{\tilde s, n}\{\{B_l\}\, l : P_l\}_{l \in L''} \rhd_{\tilde t} \Delta', \tilde s : \{\{\{A_l\}\, l : T'_l\}_{l \in L; L'} @ (\mathtt{p}, n)\}}$$

□

Proof B2 (Theorem: Subject Reduction)**.**
We prove

If $\mathsf{true}; \Gamma \vdash P \rhd_{\tilde s} \Delta$, Δ *coherent and* $P \to P'$
then $\mathsf{true}; \Gamma \vdash P' \rhd_{\tilde s} \Delta'$ *where* $\Delta \to^{0/1} \Delta'$.

By induction on the derivation of $P \to P'$.
We prove the case Sync. The remaining cases can be generated by adding assertion arguments to the proof in [2]. Assume

$$\dfrac{[\textsc{Sync}]\qquad h \in \bigcap_{i=1}^{n} L_i \quad B_{1h} \downarrow \mathsf{true} \quad \ldots \quad B_{nh} \downarrow \mathsf{true}}{\mathbf{sync}_{\tilde s, n}\{\{B_{1l}\}\, l : P_{1l}\}_{l \in L_1} \mid \ldots \mid \mathbf{sync}_{\tilde s, n}\{\{B_{nl}\}\, l : P_{nl}\}_{l \in L_n} \to P_{1h} \mid \ldots \mid P_{nh}}$$

We can assume that the typing $\mathsf{true}; \Gamma \vdash \mathbf{sync}_{\tilde t, n}\{\{B_{1l}\}\, l : P_{1l}\}_{l \in L_1} \mid \ldots \mid \mathbf{sync}_{\tilde t, n}\{\{B_{nl}\}\, l : P_{nl}\}_{l \in L_n} \rhd_{\tilde s} \Delta, \tilde t : \{\{\{A_{il}\}\, l : T_{il}\}_{l \in L; L'} @ (\boldsymbol{i}, n)\}_{i \in \{1..n\}}$ *starts with* $n-1$ *applications of the Conc rule each containing one application of the Sync rule because of the extension of Lemma 5.18 in B1. This gives us the subderivations:*

$$\dfrac{[\textsc{Sync}]\quad \begin{matrix} \forall l \in L_i.\, \dfrac{\mathcal{D}_{il}}{\mathsf{true} \wedge A_{il}; \Gamma \vdash P_{il} \rhd_{\tilde s_i} \Delta_i, \tilde t : \{T_{il} @ (\boldsymbol{i}, n)\}} \\ \forall l \in L_i.\, \vdash \mathsf{true} \Rightarrow (B_{il} \Rightarrow A_{il}) \quad \cdots \end{matrix}}{\mathsf{true}; \Gamma \vdash \mathbf{sync}_{\tilde t, n}\{\{B_{il}\}\, l : P_{il}\}_{l \in L_i} \rhd_{\tilde s_i} \Delta_i, \tilde t : \{\{A_{il}\}\, l : T_{il}\}_{l \in L'; L} @ (\boldsymbol{i}, n)}$$

for i=1..n such that $\tilde{s}_i \cap \tilde{s}_j = \emptyset$ *for all* $i \neq j$ *in* $1..n$, $\bigcup_{i=1}^{n} \tilde{s}_i = \tilde{s}$ *and* $\Delta_1 \circ (\Delta_2 \circ (\ldots \circ \Delta_n)) = \Delta$.

Since each of these subderivations starts with the Sync rule we get that

$$\frac{\mathcal{D}_{ih}}{\mathsf{true} \wedge A_{ih}; \Gamma \vdash P_{ih} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{ih}@(\boldsymbol{i}, n)\}} \quad \textit{for } i = 1..n$$

Since for each i: $B_{ih} \downarrow \mathsf{true}$ *we have that* $\vdash B_{ih}$*, and since* $\vdash \mathsf{true} \Rightarrow (B_{ih} \Rightarrow A_{ih})$ *we get that* $\vdash A_{ih}$ *by cut elimination of* B_{ih}*. Finaly by applying cut-elimination of* A_{ih} *to each proof in* $\mathcal{D}_{ih}$ *we get that*

$$\frac{\mathcal{D}'_{ih}}{\mathsf{true}; \Gamma \vdash P_{ih} \rhd_{\tilde{s}_i} \Delta_i, \tilde{t} : \{T_{ih}@(\boldsymbol{i}, n)\}} \quad \textit{for } i = 1..n$$

Now we can apply Conc $n-1$ *times to create a derivation of*
$\mathsf{true}; \Gamma \vdash P_{1h} \mid \ldots \mid P_{nh} \rhd_{\tilde{s}} \Delta, \tilde{t} : \{T_{ih}@(\boldsymbol{p}, n)\}_{\boldsymbol{p} \in \{1..n\}}$.
Since $\Delta, \tilde{t} : \{\{\{A_{il}\}\, l : T_{il}\}_{l \in L; L'}@(\boldsymbol{i}, n)\}_{i \in \{1..n\}} \to \Delta, \tilde{t} : \{T_{ih}@(\boldsymbol{i}, n)\}_{i \in \{1..n\}}$*, subject reduction is fulfilled in the Sync case.* □