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
{starcke,lnielsen,henglein}@diku.dk
[2] IT University of Copenhagen
hilde@itu.dk
[3] Imperial College London
yoshida@doc.ic.ac.uk

Abstract This paper proposes a new theory of *multiparty session types* extended with *propositional assertions* and *symmetric sum types* for modelling collaborative distributed workflows. Multiparty session types statically guarantee that workflows are type-safe and deadlock-free, facilitate automatic generation of participant-specific ("local") workflow protocols from global descriptions, and support flexible implementation of local workflows guaranteed to be compliant with the workflow protocols. The extensions with assertions and symmetric sum types support expressing *state-based (pre)conditions* and *consensual multiparty synchronisation*, which are common in complex distributed workflows.

We demonstrate the theory's applicability to *clinical practice guidelines (CPGs)* by providing a prototype implementation targeting mobile healthcare applications. It compiles declarative healthcare workflows specified in a flexible spreadsheet-formatted *process matrix* into type-checked multiparty processes. The type-checked processes are interpreted on a server communicating with generic, stateless clients running on Android tablet computers, which addresses the pervasiveness requirements common to clinical and home healthcare scenarios. A physician has, with little prior training, successfully used the prototype to design her own healthcare workflow as a process matrix, employing instantaneous test and usage feedback from the prototype.

1 Introduction

Healthcare processes are characterised by being highly mobile, collaborative, security critical, and requiring a high degree of flexibility and adaptability [3,9]. Furthermore, they typically involve complex decisions based on data collected during the process, and they are regulated, e.g. by law and clinical practice guidelines (CPGs) [25]. These characteristics make healthcare processes a particularly challenging class of case management processes [11] in need of computerised support. Their design and implementation needs to support *pervasive* execution and to be highly *trustworthy*, where formalised and verifiable process models can play a particularly important role. In the present paper we focus on how formalised models based on a compilation from a declarative process model to a new variant of multi-party session types can support pervasive execution and increase the trustworthiness. Pervasive execution is supported by automatic distribution of guideline protocols using the theory of end point projections, implemented in a prototype demonstrator allowing pervasive access to guidelines via stateless clients running on Android tablet computers. Trustworthiness is increased in two ways: 1) The declarative input format allows for specifying guidelines simply as the set of basic activities and their causal constraints instead of a procedure or flowchart. This frees the domain experts from having to "think as computers", which as stated by Parnas [20] is obviously hopeless for concurrent systems. 2) The theory of multiparty session types allows for statically guaranteeing deadlock freedom.

CPGs are descriptions of medical treatment procedures, typically maintained by professional medical associations at the national level, for specific medical disorders. CPGs can express workflows and various cooperations among healthcare processes, which are formed by the diverse collaborative patterns between multiple participants. That is, a CPG is an agreement of *global protocol* or *guideline* between distributed organisations or participants. A pattern that 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 step in the protocol. Such global protocols with symmetric, multiparty synchronisations are naturally expressed in a *choreography language*, such as the WS-CDL [27] or the BPMN 2.0 [18] choreography notation exemplified in Fig. 1 in Sec. 2.

Traditionally, workflow process models and choreographies, and also CPGs, are represented as flow-graphs inspired by and based on the seminal work on the Petri Nets model [26,13], where safety and liveness properties can be verified using model checking techniques [14]. As pointed out in [9], most of this work has been focusing on centralised models and executions of the global protocols.

In the present paper we leverage the work on *session types* and *end-point projections* [8], which provides a foundation for *decentralised execution* and *verification by type checking* of protocols in general, and CPGs in particular, specified globally as choreographies. The framework of multiparty session types provides a formal choreography model language typed with *global multiparty session types* that guarantee that well-typed processes are deadlock free and can be projected to session typed end-point processes (i.e. corresponding to BPMN processes for each participant).

The work on choreographies and session types has, however, so far been focusing on process models with explicit control flow (variations of the π-calculus), which have been observed to have limitations when it comes to flexibility and adaptability [1]. As an alternative, formal declarative process notations with implicit control flow have been proposed and investigated as a means to provide more support for adaptability in case management systems in general [1,21,5,23] and health care processes in particular [6,9].

The key contributions of the present paper are to show 1) how the theory of multiparty session types [8] extended with logical predicates [4] and symmetric sum types [16] can be used to compactly represent *declarative*, distributed, and collaborative workflows, that 2) can be modelled as a global guideline by domain experts and 3) verified for deadlock-freedom statically, i.e. at compile time, using automatic code generation and type inference, and 4) interpreted in a decentralised way to provide a pervasive execution on generic tablet clients.

Concretely we show in Sec.2 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 in Sec. 3 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. Finally we briefly outline in Sec. 4 the formal theory behind the approach and the properties it ensures, and describe related and future work in Sec. 5.

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 Sec. 2.1 we describe the example workflow as a BPMN 2.0 Choreography diagram and the corresponding Process Matrix spreadsheet. We then demonstrate in Sec. 2.2 how the process matrix workflow processes can be described compactly in multiparty session types with assertions and symmetric sum types. Finally we overview the prototype implementation in Sec. 2.3.

2.1 Example Workflow as Choreography and Process Matrix

A simple CPG workflow involving three participants is described in Fig. 1 as a Choreography 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 shown as a circle with a thin border at the left of the diagram). Then two tests, Test1 and Test2, 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 sent by the nurse to both the patient and the doctor. Each test may be repeated, as indicated by the repeating subprocess (the looping 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 (following the bottom ”ok” branch), or the doctor prescribes a drug for the patient (following the top ”not ok” branch), sending the prescription to both the patient and the nurse. The workflow is ended when the patient is discharged, indicated by the end event shown as a circle with a thick border at the right

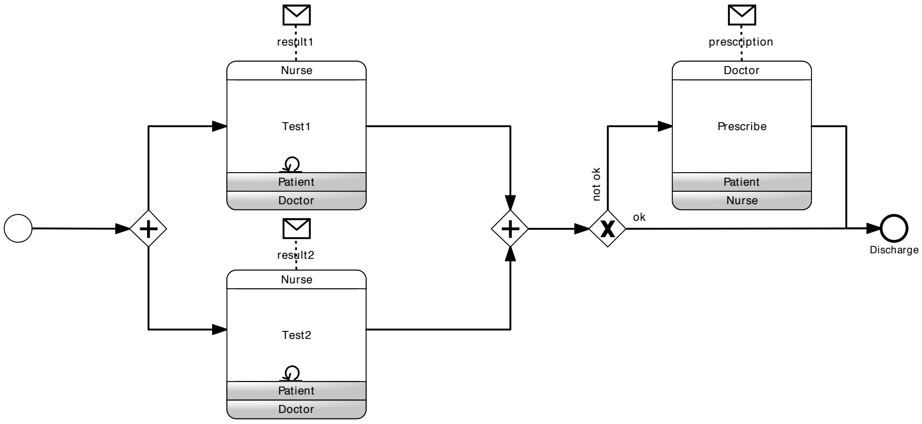


Figure 1. Workflow as BPMN 2.0 Choreography

of the diagram. 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 for 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 (http://www.resultmaker.com/) in the TrustCare research project. The process matrix corresponding to the choreography in Fig. 1 is shown in Fig. 2 below, using three boolean data fields (pre, result1, and result2) explained 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 name, access control (**R**ead or **W**rite) for each participant (**P**atient, **N**urse, **D**octor), **Seq**uential predecessor relation, **Log**ical predecessor relation (not used in our simple example), Conditions, Input data, and an optional Action performed when the activity is executed. The condition field must evaluate to true for an activity to be enabled. Actions are given in a small if-then-else language:

$$\begin{aligned}&\text{Cmd} ::= \text{c} \mid \text{end},\\ &\quad\text{c} ::= \ \text{set}(x) \mid \text{reset}(x) \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \{c_1; \ldots; c_n\},\\ &\quad\text{e} ::= x \mid \neg\, e \mid e_1 \wedge e_2 \mid e_1 \vee e_2\end{aligned}$$

where the *end* command ends the workflow, *set(x)* and *reset(x)* sets the value of the variable x to true or false respectively. For instance, when the prescription activity is executed, the pre variable is set to true, which disables the Test1 and Test2 activities. This also allows to represent non-determinism, i.e. branching behavior. If-then-else considers a Boolean expression and then uses either of the commands. The bracketed commands are performed in sequence. Furthermore, every sequential predecessor (for which the condition field presently evaluates to true) must have been executed at least once before the activity can be executed.

A logical predecessor of an activity enforces the extra constraint that if the logical predecessor is re-executed then the activity must also be re-executed. Thus, 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, looping behavior (or jumping back to previous activities) is the ”default”. In particular, Test1 and Test2 can be repeated as long as the prescription has not been made. This means that flexibility (of the worker) is the default; if the work flow is to be constrained, i.e. be less flexible, the constraints must be explicitly given. For instance, if tests should be allowed also after a prescription, and in that case requiring a new prescription if both tests are still not ok, one could simply change the matrix to the one given in Fig. 3.

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. More flexible CPG with tests being logical predecessors of prescription.

The same flexibility can of course be obtained using a choreography as the two notations are equally expressive, but in the process matrix notation, flexibility in execution is the default. Activities can be listed in the ”normal” order, but repeated by default if necessary. Also, processes can be changed incrementally e.g. by adding rows and changing conditions. Hereto comes, that spreadsheets are familiar to many users, in particular if they have used Excel. Also, it was observed in a field study, that the tabular process descriptions actually corresponded to the paper based records used at the hospitals to keep track of the treatment [12].

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 propositions as assertions and with so-called symmetric sum types.

```
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

Multiparty session types [8] define protocols for interactions in a group of participants. They closely correspond to choreographies. In addition to defining the protocol, the theory of session types guarantees type-safety and deadlock freedom.[4] Moreover, it facilitates verifying 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* [4] refines type signatures with logical predicates, which can be used to restrict the values that are communicated and choices that are made. We also use *symmetric sum types* [17], which are an extension of multiparty session types that can type nondeterministic choice agreed upon by multiple participants.

These three main features—multiparty, symmetric synchronisations and logical predicates—are essential for representing process matrix workflows in a direct and compact way and verifying practical use cases, not only in the context of CPGs, but also for workflows in general.

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), parameterised by a *state*: test1, and test2, describe if the respective test action has already been executed; this is needed because the test actions are sequential predecessors of the prescribe and discharge actions. The pre condition records whether the prescription activity has been executed. It is used to ensure prescription is executed only once and to block subsequent test1 and test2 actions. Finally, result1 and result2 record the results of the respective tests. The type is a symmetric sum (choice) with options specified by the underlined labels, Test1, Test2, Prescribe, and Discharge. The intuition is that all participants symmetrically agree on one of the four actions.

[4] This is referred to as *progress* in the theory of session types. This may be confusing, since progress is also used as synonym for liveness, i.e. that something *good* eventually happens, which is not guaranteed by the present theory of session types.

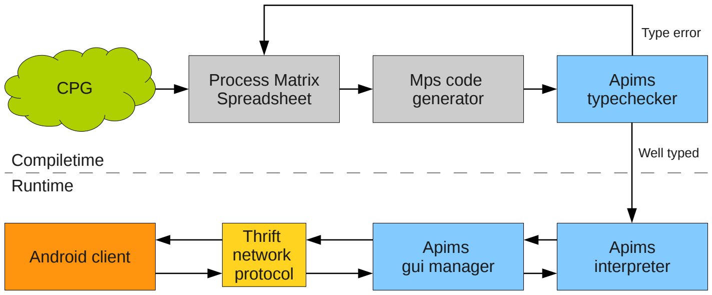


Figure 5. Demonstrator architecture.

After executing an action, the recursive type is reentered with an updated state, except if the action is Dischargȩ, which ends the workflow. In the state where both tests have been executed, no prescription has been made yet and at least one of the test results was not ok (represented as the Boolean value false), 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 enforce that doses of medicine be below a particular limit. For simplicity, this has not been included in our example, however. In the example workflow assertions are used to ensure that the same result is sent to the patient and the doctor, and control wether the medicine must be administered or the patient can be discharged directly.

2.3 Implementation

The demonstrator allows distributed execution of workflows specified by a process matrix on a server accessed by Android tablet clients. The architecture is depicted in Fig. 5. 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 value (CSV) file produced from an off-the-shelf spreadsheet program. It enables the process matrix to be specified in a normal spreadsheet program, which provides a graphical table editor familiar to many end-users.

Code generation. The generated mps code consists of a global multiparty session type, as exemplified in Fig. 4, representing the global workflow protocol, and the local process for each participant.

The code for the local processes contains user-interface information which, when interpreted, prompts users for information through a graphical user interface (GUI). The local process code generation, including its GUI-actions, is configurable; concretely, it is generated from descriptions written in a separate spreadsheet table.

Fig. 6 shows 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 indicates that this is participant 2, the doctor).

```
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.

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. In contrast to BPMN choreographies, a process matrix allows actions with more than 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 conditions derived from the workflow and for each action whether it has been executed. Each writer action receives inputs from the GUI (line 20) and sends them to the reader participants (lines 21, 22).

The predecessor and activity conditions 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 state is 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 action that has the completed action as logical predecessor is set to false.

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.

Apims. As part of our architecture, we have have created an `ASCII` syntax for the asynchronous π-calculus with multiparty sessions and symmetric synchronisation called APIMS, and implemented a type checker and an interpreter. This is to our knowledge the first prototype implementation of the π-calculus with multiparty sessions and multiparty session types. The implementation along with example programs can be found on the APIMS website [2].

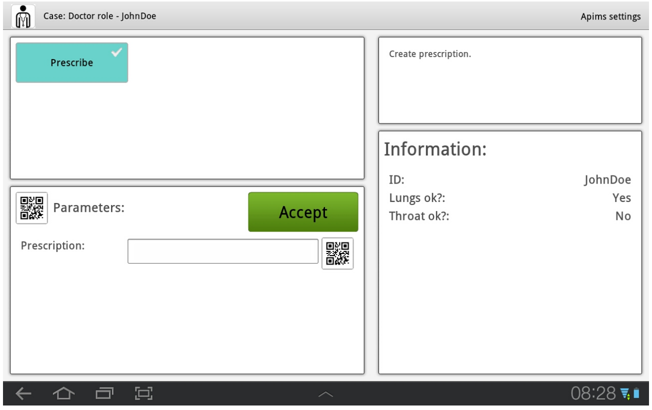
The arrow connecting the mps code generation and the apims type checker in Fig. 5 shows that the mps code is type checked with the apims type checker. If the code is not well-typed it will in this case be because the workflow may deadlock, i.e. it may reach a state that is not the final state, but no activity can be executed. The example process matrices given above produce well-typed code. An innocent-looking modification such as changing the logical or ($\vee$) in the condition for the Discharge activity to a logical and ($\wedge$) would make it possible to deadlock, however: if both tests are fine (blocking the prescription), the missing prescription prevents the discharge of the patient. The static type checking thus allows 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. [5]

If the code is well-typed, the apims interpreter in the lower right of Fig. 5 interprets the code of each participant process. It communicates with the user interfaces of the clients through a GUI manager, a separate, replaceable module that communicates with the clients and maintains a view of the global process state for each participant. In particular, each guisync term introduces a list of *choices* for each participant corresponding 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 be accepted by the clients, and if all parties accept a choice, execution can continue with the corresponding branch.

[5] However, the current implementation does not provide a very useful feedback to the non technical user.

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

These data are encoded in the key-value pair of each guivalue. Note that clients are *stateless*: By keeping all data in the interpreter, clients can be changed/break down without ruining the execution.



Case: Doctor role - JohnDoe

Create prescription.

Information:

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

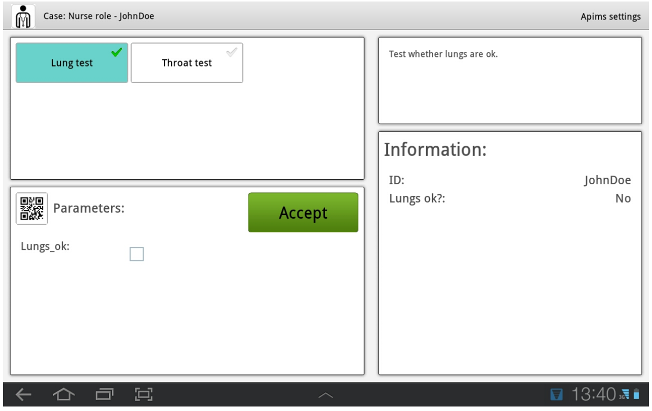
Parameters:

Prescription:

Figure 7. Screenshot from Android client logged in as Doctor.

In Fig. 6 the guivalue in line 2 assigns the Doctor role to the specific part of the code. This lets the GUI manager know which choices are assigned to which role. Fig. 7 shows a screenshot of the Android client running the example workflow in the doctor role, which can be seen at the top of the screen. The workflow is in a state where the nurse has performed both tests. The other guivalues all correspond to different parts of the screen. The guivalue in line 3 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 line 23 results in the values shown on the right-hand side once the Doctor has entered those. The last guivalue in line 24 is used to pass the execution state of the action to the client. This results in a small checkmark filled with a green colour for the action, so the user knows that it has been performed. In the screenshot the execution state is false, so the checkmark is not filled, i.e. it is shown as white.

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 only the local process corresponding to the workflow relevant to the doctor, and the Android client used by the Nurse presents only the local process relevant to 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.



Test whether lungs are ok.

Information:

ID: JohnDoe

Lungs ok?: No

Lungs_ok: □

Figure 8. Screenshot from Android client logged in as Nurse.

The model-view-controller architecture of apims supports fully flexible interface design without compromising the trustworthiness of correct execution of the specified workflows. Furthermore, the communication between clients and apims is mediated through the interface definition language framework Thrift [24], which supports multiple language bindings. Altogether, this supports flexible client design for usability in a pervasive context; e.g., a simple approach to secure and efficient inputting on a tablet computer has been by using QR-codes scanned though the tablet's camera. This enables a user to scan the drug name and the dose from physical objects, minimising the amount and attendant risks of manual typing. We have only superficially touched upon the technical and usability challenges of developing user interface clients for tablet computers in comparison to conventional PC clients, however.

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.

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 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 expressiveness 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 were 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.

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

The developed workflow is shown in Fig. 9 and Fig. 10. The roles are: Doctor (D), Nurse (N), Secretary (S), Anaesthesiologist (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).

3.2 Evaluation

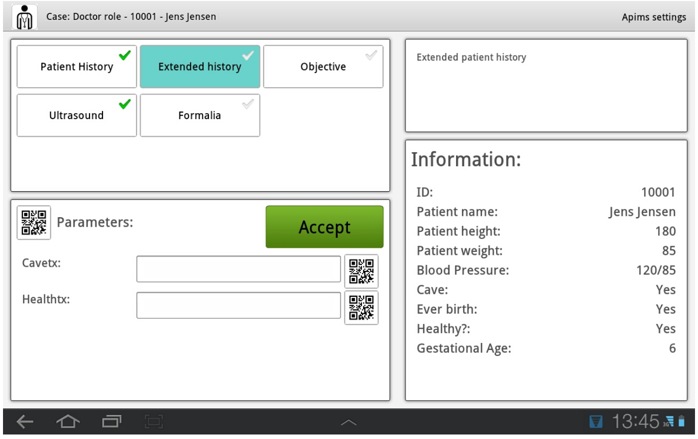
Generally the experiment turned out very successfully: DFL was easily able to use the spreadsheets to build her own workflow. The instructing author only had to take over one time to fix a problem. Even though the workflow included fairly complex logic, DFL was able to create it without any previous 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. See [16] for detailed definitions and proofs.

Once given global types as a description of global interactions among communicating processes, we can consider the following development steps for validating programs.

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



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.

In our implementation the global type is generated from a Process Matrix spreadsheet, by the mps code generator depicted in Fig. 5, i.e. the domain expert writes a spreadsheet instead of a global type.

Step 2 Projections of global type G (called *local types*) onto each participant are generated, either by a programmer or as in our implementation automatically.

Step 3 Program code P, one for the local behaviour of each participant $\mathtt{p}$, is generated and its conformance to local type T is validated by efficient type-checking. The mps code generator in the implementation actually *creates* default implementations for each participant. A programmer can use the default implementations as starting point and then develop more refined implementations, possibly in other session typed end-point languages, while adhering to the projected local type.

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

Going back to the example from Sec. 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 from the local type(Step 3). In Fig. 12 is given the local type of the patient and its behaviour described as an end-point process in the π-calculus, extended with the `sync` primitive. 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`).

The implementation extends the π-calculus with a `guisync` constructor, which is the result of extending the `sync` for user input. Each branch has a set of typed arguments that must be given using the GUI before that choice is accepted, and the given arguments can be used by the process in that branch. In Fig. 6 `guisync` is seen in line 8, and the user input can be seen as “prescription” in line 20.

```
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

We then type-check processes by following the session typing rules. The typing judgement extends the original one [4] 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. By the rules, we can verify the example is type-able, 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 the nurse implemented similarly to P_P and $\mid$ denotes parallel composition. We end this section by stating the *subject reduction theorem*, which guarantees that once the process is compiled, then there will be no type error at runtime.

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

From this theorem, we can derive many safety properties as corollaries [8, Sec. 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 is always matched to its corresponding receiving action at the same channel); (3) **session fidelity**: the interactions of a type-able process exactly follow the specification described by its global type; and (4) **progress**: once a communication has been established, well-typed programs will never get stuck at communication points. The formal definitions and the proofs of these properties can be found in [16,4,8].

5 Conclusions, Related and Future Work

We have successfully applied the symmetric sum types and assertion extensions of the multiparty session types to compactly specify flexible, declarative workflows with data constraints as needed for CPGs. This enables a decentralised execution automatically generated from the specification, which is guaranteed to be deadlock free by type checking and the subject reduction theorem. We provided an end-to-end, model-driven, pervasive demonstrator implementation. Finally, we reported on a successful experiment, letting a physician declaratively specify her own CPG in an off-the-shelf spreadsheet program and run it on the demonstrator.

The original implementation of the Process Matrix called *Online Consultant* by *Resultmaker* [12] is database based. This means that communication consists of the sender uploading information to the server, and all participants must query the server when using the information. Implementing the workflows based on the π-calculus and session types not only gives the Process Matrix a formal semantics, but also allows an implementation where participants communicate their data as peer-to-peer. This offers more natural and robust realisation of the workflows. It is important to point out that while the current demonstrator prototype executes all participant threads on a single, central server, there is nothing that hinders decentralised execution of such threads either on local servers or even at the clients. However, if executed (only) on a mobile client one loses the possibility to access/recover the thread if the mobile client is lost or damaged. Also, the theorem provers we have implemented (to verify assertions in the type checker) are based on the LK and CFLKF proof systems, which are not very efficient in practice. But there is an abundance of theorem provers available [22,10] 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 approach in the present paper relates to work based on the Lightweight Coordination Calculus (LCC) [23,9] in being decentralised and representing clinical protocols and guidelines as message-based interaction models, which exchange information among agents distributed across different hospitals. As pointed out in [9], most other approaches ([19]) providing formal modelling, enactment and verification of CPGs have been based on centralised models and executions. While the work based on LCC focuses only on describing the individual agents, the approach based on global and local session types taken in the present paper combines the best of both worlds: the global session type corresponds to the centralised, global overview of the CPG and the local session types generated automatically from the global session type provide the individual views. Moreover, instead of relying on model checking (of the combined system of agents) the session type approach extracts the individual communication protocols which can be type-checked against an individual agent thread implemented in the π-calculus being interpreted in the current demonstrator. This also opens up possibilities for implementing local session type checking for other end-point languages, such as Java, Python, C, Ocaml, LCC and related formal notations described below.

Another related approach is the declarative Dynamic Condition Response (DCR) Graphs process model [5,15] developed in the TrustCare project. DCR

Graphs can be verified for safety and liveness properties using the SPIN model checker [15] and directly formalise the declarative process matrix model and extend it with the possibility of differentiating between may and must behaviours, that is, an activity may be possible, but not required in order to fulfil the goal of the workflow. As for the session types approach, DCR Graphs allow global descriptions from which end-point descriptions can be derived automatically and executed locally [7,6]. The distribution technique is more flexible than the one based on session types, since it is not restricted to a fixed allocation of participants in the global description. However, DCR Graphs have so far limited support for data and no facility for type and assertion checking for local agents as developed in this paper. This makes the DCR Graphs approach less flexible with respect to end point implementations which must be based on the projected DCR Graphs.

As for future work, it would indeed be interesting to explore session typed LCC, Petri Nets and DCR Graphs, which would enable to provide end-points in these languages and to include and benefit from the work on these alternative approaches. We also plan to explore an extension of session types with time deadlines and violations of such, which are crucial in order to represent real CPGs. Finally, we are planning a larger experiment with several users and CPGs in collaboration with It, Medico og Telefoni (IMT) (www.regionh.dk).

Acknowledgements This work was funded in part by the Danish Council for Strategic Research, Grant #2106-07-0019, the IT University of Copenhagen and University of Copenhagen (the TrustCare project, www.trustcare.eu). We want to thank Dorthe Furstrand Lauritzen for participating in the experiment and for extensive feedback on the demonstrator, and the anonymous reviewers for their careful reviews and comments.

References

1. W. M. P. v. d. Aalst, M. Pesic, and H. Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D*, 23(2):99–113, 2009.
2. Apims project page. http://www.thelas.dk/index.php/apims.
3. J. E. Bardram and C. Bossen. Mobility work: The spatial dimension of collaboration at a hospital. *CSCW*, 14:131–160, 2005.
4. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, LNCS, pages 162–176. Springer, 2011.
5. T. Hildebrandt and R. R. Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. In *Proceedings of PLACES'10*, 2011.
6. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Declarative modelling and safe distribution of healthcare workflows. In *Foundations of Health Informatics Engineering and Systems (FHIES)*, volume 7151 of *LNCS*, pages 39–56, Berlin, Heidelberg, August 2011. Springer-Verlag.
7. T. Hildebrandt, R. R. Mukkamala, and T. Slaats. Safe distribution of declarative processes. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 237–252, Berlin, Heidelberg, 2011. Springer-Verlag.

8. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
9. B. Hu, S. Dasmahapatra, D. Robertson, and P. Lewis. Decentralised clinical guidelines modelling with lightweight coordination calculus. In *LBM*, December 2007.
10. J. A. Kalman. *Automated reasoning with Otter.* Rinton Press, 2001.
11. J. Koehler, J. Hofstetter, and R. Woodtly. Capabilities and levels of maturity in it-based case management. In *Proceedings of the 10th international conference on Business Process Management*, BPM'12, pages 49–64, Berlin, Heidelberg, 2012. Springer-Verlag.
12. K. Lyng, T. Hildebrandt, and R. Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *ProHealth '08*, 2008.
13. W. MacCaull and F. Rabbi. Nova workflow: A workflow management tool targeting health services delivery. In *Foundations of Health Informatics Engineering and Systems (FHIES)*, volume 7151 of *LNCS*, pages 74–91, Berlin, Heidelberg, August 2011. Springer-Verlag.
14. W. MacCaull, F. Rabbi, and A. Mashiyat. Model checking workflow monitors and its application to a pain management process. In *Foundations of Health Informatics Engineering and Systems (FHIES)*, volume 7151 of *LNCS*, pages 110–127, Berlin, Heidelberg, September 2011. Springer-Verlag.
15. R. R. Mukkamala. *A Formal Model For Declarative Workflows - Dynamic Condition Response Graphs.* PhD thesis, IT University of Copenhagen, 2012.
16. L. Nielsen. *Regular Expressions and Multiparty Session Types with Applications to Workflow Based Verification of User Interfaces.* PhD thesis, University of Copenhagen, 2012.
17. L. Nielsen, N. Yoshida, and K. Honda. Multiparty symmetric sum types. In *EXPRESS'10*, volume 41 of *EPTCS*, pages 121–135, 2010.
18. Object Management Group BPMN Technical Committee. Business Process Model and Notation, version 2.0. Webpage, January 2011. http://www.omg.org/spec/BPMN/2.0/PDF.
19. Open clinical. guideline modelling methods summaries. Webpage. www.openclinical.org/gmmsummaries.html.
20. D. L. Parnas. Software aspects of strategic defense sytsems. *Communications of the ACM*, 28(12):1326–1335, 1985. Reprinted from Journal of Sigma Xi, Vol. 73. No. 5. pp. 432-440.
21. M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *Proceedings of the 2006 international conference on Business Process Management Workshops*, BPM'06, pages 169–180, Berlin, Heidelberg, 2006. Springer-Verlag.
22. A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI communications*, 15(2, 3):91–110, 2002.
23. D. Robertson. A lightweight coordination calculus for agent systems. In *Declarative Agent Languages and Technologies*, pages 183–197, 2004.
24. M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Available from http://thrift.apache.org/.
25. A. ten Teije, S. Miksch, and P. Lucas. *Computer-based Medical Guidelines and Protocols: A Primer and Currend Trends.* Studies in Health Technology and Informatics. IOS Press, 2008.
26. W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
27. Web Services Choreography Working Group. Choreography Description Language. http://www.w3.org/2002/ws/chor/.