SOCA (2015) 9:269–284
DOI 10.1007/s11761-014-0172-8

SPECIAL ISSUE PAPER

Pabble: parameterised Scribble

Nicholas Ng · Nobuko Yoshida

Received: 12 March 2014 / Revised: 14 November 2014 / Accepted: 16 November 2014 / Published online: 20 December 2014

Abstract Many parallel and distributed message-passing programs are written in a parametric way over available resources, in particular the number of nodes and their topologies, so that a single parallel program can scale over different environments. This article presents a parameterised protocol description language, Pabble, which can guarantee safety and progress in a large class of practical, complex parameterised message-passing programs through static checking. Pabble can describe an overall interaction topology, using a concise and expressive notation, designed for a variable number of participants arranged in multiple dimensions. These parameterised protocols in turn automatically generate local protocols for type checking parameterised MPI programs for communication safety and deadlock freedom. In spite of undecidability of endpoint projection and type checking in the underlying parameterised session type theory, our method guarantees the termination of end point projection and type checking.

Keywords Multiparty session types · Communication safety · Deadlock freedom · Pabble · Protocol description language · Dependent types

1 Introduction

Message-passing is becoming a dominant programming model, as witnessed in application programs from high-performance computing scaling over thousands of cores or cloud-based scalable backends of popular web services. These are environments where services are dynamically provided, through choreography of interactions among numerous distributed components. Assuring safety of concurrent software in these environments is a vital concern: Many message-passing libraries, programs and systems are shared and long-lived, and some process sensitive data, so that safety violations such as deadlocks and incompatible messaging patterns or data payloads between senders and receivers can have catastrophic and unexpected consequences [10].

Our proposal for safety assurance for message-passing programs is based on *multiparty session types* [13]. The methodology considers the specification of a global interaction protocol among multiple participants, from which we can derive a local protocol for an individual participant. Once each program is type-checked against its local protocol, a set of typed programs is guaranteed to run without deadlock or communication mismatches. We based our work on [20], where the authors proposed a programming framework for message-passing parallel algorithms, centering on explicit, formal description of global protocols, and examined its effectiveness through an implementation of a toolchain for the C language. The toolchain uses a language Scribble [12,24] for describing the multiparty session types in a Java-like syntax. A simple example of a protocol in Scribble which represents a ring topology between four workers is given below:

```
1  global protocol Ring(role Worker1, role
       Worker2, role Worker3, role Worker4){
2    rec LOOP {
3      Data(int) from Worker1 to Worker2;
4      Data(int) from Worker2 to Worker3;
```

N. Ng (✉) · N. Yoshida
Department of Computing, Imperial College London,
London SW7 2AZ, UK
e-mail: nickng@imperial.ac.uk

N. Yoshida
e-mail: n.yoshida@imperial.ac.uk

```
5       Data(int) from Worker3 to Worker4;
6       Data(int) from Worker4 to Worker1;
7       continue LOOP;
8     }
9   }
```

A Scribble protocol starts from the keyword **global protocol**, followed by the protocol name, `Ring`. The role declarations are then passed as parameters of the protocol, which are `Worker1` through to `Worker4`. The `Ring` protocol describes a series of communications in which `Worker1` passes a message of type `Data(int)` to `Worker4` by forwarding through `Worker2` and `Worker3` in that order and receives a message from `Worker4`. It is easy to notice that explicitly describing all interactions among distinct roles is verbose and inflexible: For example, when extending the protocol with an additional role `Worker5`, we must rewrite the whole protocol. On the other hand, we observe that these worker roles have identical communication patterns which can be logically grouped together: $\texttt{Worker}_{i+1}$ receives a message from $\texttt{Worker}_i$ and the last `Worker` sends a message to $\texttt{Worker}_1$. In order to capture these replicable patterns, we introduce an extension of Scribble with dependent types called *Parameterised Scribble* (Pabble). In Pabble, multiple participants can be grouped in the same role and indexed. This greatly enhances the expressive power and modularity of the protocols. Here 'parameterised' refers to the number of participants in a role that can be changed by parameters.

The following shows our ring example in the syntax of Pabble.

```
1  global protocol Ring(role Worker[1..N]) {
2    rec LOOP {
3      Data(int) from Worker[i:1..N-1] to
           Worker[i+1];
4      Data(int) from Worker[N] to Worker[1];
5      continue LOOP;
6    }
7  }
```

role `Worker[1..N]` declares workers from `1` to an arbitrary integer `N`. The `Worker` roles can be identified individually by their indices, for example, `Worker[1]` refers to the first and `Worker[N]` refers to the last. In the body of the protocol, the sender, `Worker[i:1..N-1]`, declares multiple `Workers`, bound by the bound variable `i`, and iterates from `1` to `N-1`. The receivers, `Worker[i+1]`, are calculated on their indices for each instances of the bound variable `i`. The second line is a message sent back from `Worker[N]` to `Worker[1]`.

```
1  local protocol Ring at Worker[1..N](role
       Worker[1..N]) {
2    rec LOOP {
3      if Worker[i:2..N]   Data(int) from
           Worker[i-1];
4      if Worker[i:1..N-1] Data(int) to Worker[
           i+1];
5      if Worker[1]        Data(int) from
           Worker[N];
6      if Worker[N]        Data(int) to  Worker
           [1];
7      continue LOOP;
8    }
9  }
```

The above code shows the local protocol of `Ring`, projected with respect to the parameterised `Worker` role. The projection for a parameterised role, such as `Worker[1..N]`, will give a parameterised local protocol. It represents multiple end points in the same logical grouping.

Challenges The main technical challenge for the design and implementation of parameterised session types is to develop a method to automatically project a parameterised global protocol to a parameterised local protocol ensuring termination and correctness of the algorithm.

Unfortunately, as in the indexed dependent type theory in the λ-calculus [2,33], the underlying parameterised session type theory [9] has shown that the projection and type checking with general indices are undecidable. Hence, there is a tension between termination and expressiveness to enable concise specifications for complex parameterised protocols.

Our main approach to overcome these challenges is to make the theory more practical by extending Scribble with index notation originating from a widely used text book for modeling concurrent Java [16]. For example, notations `Worker[i:1..N-1]` and `Worker[j+i]` in the Ring protocol are from [16]. Interestingly, this compact notation is not only expressive enough to represent representative topologies ranging from parallel algorithms to distributed web services, but also offers a solution to cope with the undecidability of parameterised multiparty session types.

1.1 Overview

Figure 1 shows the relationships between the three layers: global protocols, local protocols and implementations. (1) A programmer first designs a global protocol using Pabble. (2) Then, our Pabble tool automatically projects the global protocol into its local protocols. (3) The programmer then either implement the parallel application using the local protocol as specification, or type-check existing parallel applications against the local protocol. If the communication interaction patterns in the implementations follow the local protocols generated from the global protocol, this method automatically ensures deadlock-free and type-safe communication in the implementation. In this work, we focus on the design and implementation of the language for describing parallel

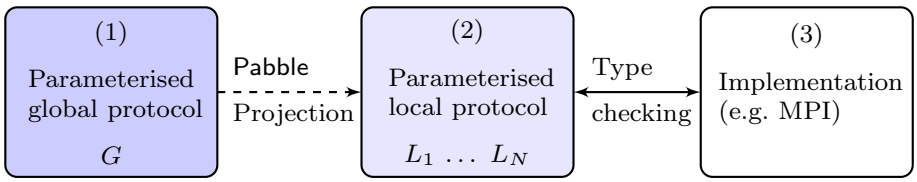


Fig. 1 Pabble programming workflow

message-passing-based interaction as global and local protocols in (1) and (2) and outline how a Pabble local type checker for MPI (3) can be implemented.

This article presents a full version of the work published in [19] which had a particular focus on modeling and expressing communication topologies in parallel applications. Apart from including the detailed proofs for the well-formedness conditions and a number of additional examples, we include use cases from web services and large-scale distributed cyber-infrastructures to show the flexibility of the Pabble language for compact parametric protocols outside the field of high-performance parallel applications. We also expand the related work for a more thorough survey and discussion on formal verification with MPI-based parallel applications.

The contributions of this article are:

- The first design and implementation of parameterised session types in a global protocol language (Pabble) (Sect. 2.2). The protocols can represent complex topologies with arbitrary number of participants, enhancing expressiveness and modularity for practical message-passing parallel programs.
- The projection algorithm for Pabble to check the well-formedness of parameterised global protocols (Sects. 2.3 and 2.4) and to generate parameterised local protocols from well-formed parameterised global protocols (Sect. 2.5). A correctness and termination proof of the projection algorithm is also presented (Sect. 2.7).
- A number of Pabble use cases in parallel programming and web services in Sect. 3.

Additional use cases of Pabble such as common interaction patterns for high-performance computing described in Dwarfs [1] can be found on the project web page [21]. We also outline a methodology for type checking source code written with MPI against Pabble protocols in Sect. 4.

2 Pabble: parameterised Scribble

Scribble [24] is a developer-friendly notation for specifying application-level protocols based on the theory of multiparty session types [3,13]. This section introduces an evolution of Scribble with parameterised multiparty session types (Pabble) defines its end point projection and proves its correctness.

2.1 The Pabble protocol language

The core elements of a Pabble protocol are interaction statements, choices and iterations. These are features common also to the Scribble language, which Pabble is extended from. Hence, Scribble protocols are compatible with Pabble, but the most expressive features such as role parameterisation can only be found in Pabble.

Interaction statements describe the messages passed between distributed participants of the protocol. For example, in the `Ring` protocol below, `Data(int)` **`from`** `Worker[1]` **`to`** `Worker[2];` is an interaction statement which sends a message from participant (called a *role*) `Worker[1]` to another participant `Worker[4]`. The participant are declared in the protocol as arguments of the protocol, **`role`** `Worker[1..4]`. The subscripting notation of the roles are for indexing the participants and will be explained in details the next section. The message has a *label*, `Data`, which may be omitted from the interaction statement. The message also contains a type name as parameters to the label, e.g., `int`, called a *payload type*. The payload type represents the data type of the message being sent.

```
1 global protocol Ring(role Worker[1..4]){
2   rec LOOP {
3     Data(int) from Worker[1] to Worker[2];
4     Data(int) from Worker[2] to Worker[3];
5     Data(int) from Worker[3] to Worker[4];
6     Data(int) from Worker[4] to Worker[1];
7     continue LOOP;
8   }
9 }
```

Choice statements are written as

```
1 choice at role {
2   Choice0() from role to roleOther;
3 } or {
4   Choice1() from role to roleOther;
5 }
```

where each of the branches is an alternative interaction subpattern which the participants can collectively select. The deciding role sends a label (e.g., `Choice0`) to other roles involved with the choice to notify them of the branch taken.

Global Pabble

```
global protocol str(para) { G }
```

Parameter

para	::=	`role` $R_d, \ldots,$ `group` str=`{` $R_d, \ldots$ `}`, $\ldots$	Role and group declaration

Global protocol body

G	::=	$l(T)$ `from` R `to` R;	Interaction
	|	`choice at` R `{` G_1 `}` `or` $\ldots$ `or` `{` G_N `}`	Choice
	|	`foreach` (b) `{` G `}`	Foreach
	|	`allreduce` $op_c(T)$;	Reduction
	|	`rec` l `{` G `}`	Recursion
	|	`continue` l;	Continue
	|	G G	Sequential composition

Payload type

T	::=	`int` | `float` | ...	Data types

Expression

e	::=	e *op* e | *num*	Binary expressions, integers
	|	$i, j, k, \ldots$ | `N`	Variables, constants
op	::=	op_c | `-` | `/` | `%` | `<<` | `>>` | `log` | ...	Binary operations
op_c	::=	`+` | `*` | ...	Commutative operations

Role

R_d	::=	*str*	Role declaration
	|	*str*`[`e`..`e`]`...`[`e`..`e`]`	Parameterised role declaration
R	::=	*str*	Roles
	|	*str*`[`h`]`...`[`h`]`	Parameterised roles
	|	`All`	*All* group role
b	::=	i : e..e	Role parameters (binding range)
h	::=	b | e	Role parameters (expressions)

Local Pabble

```
local protocol str at R_d(para) { L }
```

Local protocol body

L	::=	[`if` R] $l(T)$ `from` R;	(Conditional) Receive
	|	[`if` R] $l(T)$ `to` R;	(Conditional) Send
	|	`choice at` R `{` L_1 `}` `or` $\ldots$ `or` `{` L_N `}`	Choice
	|	`foreach` (b) `{` L `}`	Foreach
	|	`allreduce` $op_c(T)$;	Reduction
	|	`rec` l `{` L `}`	Recursion
	|	`continue` l;	Continue
	|	L L	Sequential composition

Fig. 2 Pabble syntax

Iterations (loops) in the interaction patterns are written as recursion blocks (**`rec`** `Label { }`), with **`continue`** `Label;` statement to jump back to beginning of recursion.

2.2 Syntax of Pabble

2.2.1 Global protocols

Figure 2 lists the core syntax of Pabble, which consists of two protocol declarations, global and local. A global protocol is declared with the protocol name (*str* denotes a string) with role and group parameters followed by the body G. Role R is a name with argument expressions. The argument expressions are ranges or arithmetic expressions h, and the number of arguments corresponds to the dimension of the array of roles: for example, `Worker[1..4][1..2]` denotes a 2D array with size 4 and 2 in the two dimensions, respectively, forming a 4-by-2 array of roles.

Declared roles can be grouped by specifying a named group using the keyword **`group`**, followed by the group name and the set of roles. For example,

```
group EvenWorker={Worker[2][2], Worker
    [4][2]}
```

creates a group which consists of two `Workers`. A special built-in group, **`All`**, is defined as *all processes in a session*. We can encode collective operators such as many-to-many and many-to-one communication with **`All`**, which will be explained later.

Apart from specifying ranges explicitly, ranges can also be specified using expressions. Expression e consists of the usual operators for numbers, logarithm, left and right logi-

cal shifts (<<, >>), numbers, variables (i, j, k), and constants (`M`, `N`). Constants are either *bound* outside the protocol declaration or are left *free* (unbound) to represent an arbitrary number. As in [16], when the constants are bound, they are declared by numbers outside the protocol, e.g., **const** `N = 100` or lower and upper bounds, e.g., **const** `N = 1..1000`. We also allow leaving the declaration *free* (unbound), e.g., **const** `N`, as a shorthand to represent an arbitrary constant with lower and upper bounds `0` and `max`, respectively, i.e., **const** `N = 0..max`, where `max` is a special value representing the maximum possible value or practically unbounded. Binding range expression b takes the form of $i : e_1..e_n$ which means i is ranged from e_1 to e_n. Binding variables always bind to a range expression and not individual values. We shall explain the use of binding range expressions later in more details.

In a global protocol $G, l(T)$ **from** R_1 **to** R_2 is called an *interaction statement*, which represents passing a message with label l and type T from one role R_1 to another role R_2. R_1 is a *sender role* and R_2 is a *receiver role*. **choice at** R $\{G_1\}$ **or** ...**or** $\{G_n\}$ means the role R will select one of the global types $G_1, \ldots, G_n$. **rec** l $\{G\}$ is recursion with the label l which declares a label for **continue** l statement. **foreach** $(b)\{G\}$ denotes a for-loop whose iteration is specified by b. For example, **foreach** $(i : 1..n)\{G\}$ represents the iteration from 1 to n of G where G is parameterised by i.

Finally, **allreduce** $op_c(T)$ means all processes perform a distributed reduction in value with type T with the operator op_c (like **MPI_Allreduce** in MPI). It takes a mandatory predefined operator op_c where op_c must be a commutative and associative arithmetic operation. Pabble currently supports sum and product.

We allow using simple expressions (e.g., `Worker[i:0..2*N-1]`) to parameterise ranges. In addition, indices can also be calculated by expressions on bound variables (e.g., `Worker[i+1]`) to refer to relative positions of roles.

These restrictions on indices such as bound variables and relative indices calculations ensure termination of the projection algorithm and type checking. The binding conditions are discussed in the next subsection.

2.2.2 Local protocols

Local protocol L consists of the same syntax of the global type except the input from R (receive) and the output to R (send). The main declaration

local protocol *str* **at** $R_e(\ldots)\{L\}$

means the protocol is located at role R_e. We call R_e *the endpoint role*. In Pabble, multiple local protocol instances can reside in the same parameterised local protocol. This is because each local protocol is a local specification for a participant of the interaction. Where there are multiple participants with a similar interaction structure that fulfill the same *role* in the protocol, such as the `Workers` from our `Ring` example from the introduction, the participants are grouped together as a single parameterised role. The local protocol for a collection of participants can be specified in a single parameterised local protocol, using *conditional statements* on the role indices to capture edge cases. For example, in a general case of a pipeline interaction, all participants receives from a neighbor and send to another neighbor, except the first participant which initiates the pipeline and is only a sender and the last participant which ends the pipeline and does not send. In these cases, we use conditional statements to guard the input or output statements. To express conditional statements in local protocols, **if** R may be prepended to input or output statement. **if** R input/output statement will be ignored if the local role does not match R. More complicated matches can be performed with a parameterised role, where the role parameter range of the condition is matched against the parameter of the local role. For example, **if** `Worker[1..3]` will match `Worker[2]` but not `Worker[4]`. It is also possible to bind a variable to the range in the condition, e.g., **if** `Worker[i:1..3]`, and `i` can be used in the same statement.

2.3 Well-formedness conditions: index binding

As Pabble protocols include expressions in parameters, a valid Pabble protocol is subject to a few well-formedness conditions. Below, we show the conditions which ensure indices used in roles are correctly bounded. We use fv/bv to denote the set of free/bound variables defined as $\mathsf{fv}(i) = \{i\}$, $\mathsf{fv}(\mathtt{N}) = \mathsf{fv}(num) = \emptyset$ and $\mathsf{fv}(i : e_1 \ldots e_n) = \cup\mathsf{fv}(e_j)$ and $\mathsf{fv}(\textbf{foreach}\,(b)\,\{G\}) = (\mathsf{fv}(b) \cup \mathsf{fv}(G))\backslash\mathsf{bv}(b)$ and $\mathsf{bv}(i : e_1 \ldots e_n) = \{i\}$. Others are inductively defined.

1. In a global protocol role declaration, which are parameters of **global protocol**, indices outside of declared range are invalid, for example, a role `Worker[0]` is invalid if the role is declared **role** `Worker[1..3]`.
2. Let **foreach**(b_1) { **foreach**(b_2) { ...**foreach**$(b_n)\{G\}\}\}$ with $n \geq 0$:
   (a) Suppose an interaction statement $l(T)$ **from** R_1 **to** R_2; appears in G. Let $R_1 = Role_1[h_1]\ldots[h_n]$ and $R_2 = Role_2[e'_1]\ldots[e'_m]$ (we assume $n = 0$ (resp. $m = 0$) if R_1 (resp. R_2) is either a single participant or group).
      (1) $n = m$ (i.e., the dimensions of the parameters are the same)
      (2) $\mathsf{fv}(h_j) \subseteq \cup\mathsf{bv}(b_i)$ (i.e., the free variables in the sender roles are bound by the for-loops).

(3) $\mathsf{fv}(e'_j) \subseteq (\cup \mathsf{bv}(b_i)) \cup \mathsf{bv}(h_j)$ (i.e., the free variables in the receiver roles are bound by either the for-loops or sender roles);

(b) Suppose a choice statement **choice at** R { G_1 } **or** { G_2 } appears in G. Then, R is a single participant, i.e., either *Role* or *Role*[e] with $\mathsf{fv}(e) \subseteq (\cup \mathsf{bv}(b_i))$.

Condition 2(a)(1) ensures the number of sender parameters matches the number of receiver parameters. For example, the following is invalid:

$l(T)$ **from** `R[i:1..N-1][j:1..N]` **to** `R[i+1];`

Condition 2(a)(2) ensures variables used by a sender are declared by the enclosing for-loops.
Condition 2(a)(3) makes sure the receiver parameter at the j-th position is bound by the for-loops or the sender parameter at the j-th position (and not binders at other positions). For example, the following is valid:

$l(T)$ **from** `R[i:1..N-1][j:1..N]` **to** `R[i+1][j];`

But with the index swapped, it becomes invalid:

$l(T)$ **from** `R[i:1..N-1][j:1..N]` **to** `R[j][i+1];`

Condition 2(b) is similar for the case of **choice** statements where R should be a single participant to satisfy the unique sender condition in [6,8].

2.4 Well-formedness conditions: constants

In Pabble protocols, constants can be defined by

(1) A single numeric value (**const** `N=4`); or
(2) Lower and upper bound constraints not involving `max` (**const** `N=1..1000`).

Lower and upper bound constraints are designed for runtime constants, e.g., the number of processes spawned in a scalable protocol, which is unknown at design time and will be defined and immutable once the execution begins. To ensure, Pabble protocols are communication-safe in all possible values of constants, we must ensure that all parameterised role indices stay within their declared range. Such conditions prevent sending or receiving from an invalid (non-existent) role which will lead to communication mismatch at runtime.

In case (1), the check is trivial. In case (2), we require a general algorithm to check the validity between multiple constraints appeared in the regions. First, we formulate the constraints of the values of the constants as a series of linear inequalities. We then combine the linear inequalities and determine the feasible region using standard linear programming. The feasible region represents the pool of possible values in any combination of the constraints. The following explains how to determine whether the protocol will be valid for all combinations of constants:

```
1 const M = 1..3;
2 const N = 2..5;
3 global protocol P(role R[1..N]) {
4   T from R[i:1..M] to R[i+1];
5 }
```

The basic constraints from the constants are:

$$1 \leq \mathtt{M}, \mathtt{M} \leq 3, 2 \leq \mathtt{N} \text{ and } \mathtt{N} \leq 5$$

We then calculate the range of `R[i+1]` as `R[2..M+1]`. Since the objective is to ensure that the role parameters in the protocol body (i.e., `1..M` and `2..M+1`) stay within the bounds of `1..N`, we define a constraint set to be:

$$1 \leq 1 \,\&\, \mathtt{M} \leq \mathtt{N} \text{ and } 1 \leq 2 \,\&\, \mathtt{M}+1 \leq \mathtt{N}$$

which are lower and upper bound inequalities of the two ranges. From them, we obtain this inequality as a result:

$$\mathtt{M}+1 \leq \mathtt{N}$$

By comparing this against the basic constraints on the constants, we can check that not all outcomes belong to the regions, and thus, this is not a communication-safe protocol (an example of a unsafe case is $\mathtt{M} = 3$ and $\mathtt{N} = 2$). On the other hand, if we alter Line 4 to `T from R[i:1..N-1] to R[i+1];`, the constraints are unconditionally true, and so, we can guarantee all combinations of constants `M` and `N` will not cause communication errors.

Arbitrary constants In addition to constant values and lower and upper bound constants, we also consider the use cases when the value of a constant can be any arbitrary value in the set of natural numbers. This is an extension of case (2) with the `max` keyword, where we write **const** `N = 0..max` to represent a range without upper bound.

In order to check that role indices are valid with unbounded ranges, we enforce two simple restrictions. First, only one constant can be defined with `max` in one global protocol. Secondly, when the index is unbounded, its range calculation only uses addition or subtraction on integers (e.g., `i+1`).

A protocol with an invalid use of arbitrary constants is shown below:

```
1 const N = 1..max;
2 global protocol Invalid(role R[1..N]) {
3   T from R[i:1..N-1] to R[i+1];
4   T from R[j:1..N] to R[j+1];
5 }
```

If `N` is instantiated to `1`, then the role is declared to be `R[1..1]`. In the first interaction statement, `R[i:1..1-1]` is invalid, as `R[0]` is not in the range of `R[1..0]`. In the second statement, `R[j+1]` is also invalid, as it evaluates to `R[N+1]` and is out of range `R[1..N]`.

On the other hand, the following protocol is valid since the indices always stay between 0 and N.

```
1  const N = 1..max;
2  global protocol Valid(role R[0..N]) {
3    T from R[i:0..N-1] to R[i+1];
4    T from R[j:1..N] to R[j-1]; }
```

We have shown in [21], most of representative topologies with the arbitrary number of participants can be represented under these conditions.

2.5 Endpoint projection

In the next step, a Pabble protocol should be *projected* to a local protocol, which is a simplified Pabble protocol as viewed from the perspective of a given end point. The projection algorithm is explained below. To begin with the header of the global protocol

`global protocol` *name(param) { G }*

is projected onto

`local protocol` *name* **`at`** R_e*(param) { L }*

where the protocol name *name* and parameters *param* are preserved and the endpoint role R_e is declared.

Table 1 shows the projection of the body of global protocol G onto $\boldsymbol{R}$ at endpoint role R_e. The projection rules will be applied from top to bottom in the table, if a global protocol matches multiple rules, then there will be more than one line of projected protocol for a single global protocol. In Rules 1–4, we show the rule for the single argument as the same rule is applied to n-arguments. Each rule is applied if $\boldsymbol{R}$ meets the condition in the second column under the constraints given by the constant declarations. Rules 1 and 2 show the projection of the interaction statement when $\boldsymbol{R}$ appears in the receiver and the sender position, respectively. Since $\boldsymbol{R}$ is a single participant, it should satisfy $\boldsymbol{R} = R_e$ (i.e., the role is the endpoint role). The projection simply removes the reference to role $\boldsymbol{R}$ from the original interaction statement.

Rules 3 and 4 show the projection of an interaction statement if role $\boldsymbol{R}$ is a parameterised single participant where $\boldsymbol{R}$ is an element of the endpoint role R_e. For example, if R_e = `Worker[1..3]`, $\boldsymbol{R}$ can be either `Worker[1]`, `Worker[2]` or `Worker[3]`. In addition to removing the reference of role $\boldsymbol{R}$ in the receive and send statements, we also prepend the conditions which the role applies. The order of which the projection rules are applied ensure that an interaction statement will be localized to receive then send. In general, both receive–send or send–receive in the projected local protocol are correct, as long as the projection algorithm is consistent and the well-formedness conditions of the global protocol are satisfied. The global protocol will ensure, by session typing, that a send will have a matching receive at the same stage of the protocol.

Rule 5 is for all-to-all communication. Any role $\boldsymbol{R}$ will send a message with type U to all other participants and will receive some value with type U from all other participants. Since all participants start by first sending a message to all, no participant will block waiting to receive in the first phase, so no deadlock occurs.

Rules 6 and 7 are the projection rules for the case that we project onto a group. We need to check that a group is a subset of the endpoint role R_e with respect to the group declarations in the global protocol. Then, the rules can be understood as Rules 3 and 4.

Rules 8 and 9 show the projection of interaction statements with parameterised roles using relative indexing (we show

Table 1 Projection of G onto $\boldsymbol{R}$ at the end point role R_e

		Conditions	Global protocol	Local protocol projected onto $\boldsymbol{R}$ at R_e
1.	Receive	$\boldsymbol{R} = R_e$	U `from` R' `to` $\boldsymbol{R}$	U `from` R'
2.	Send	$\boldsymbol{R} = R_e$	U `from` $\boldsymbol{R}$ `to` R'	U `to` R'
3.	Receive (parametric)	$\boldsymbol{R} \in R_e$	U `from` R' `to` $\boldsymbol{R}$	`if` $\boldsymbol{R}$ U `from` R'
4.	Send (parametric)	$\boldsymbol{R} \in R_e$	U `from` $\boldsymbol{R}$ `to` R'	`if` $\boldsymbol{R}$ U `to` R'
5.	All to All		U `from All to All`	U `to All` ; U `from All`
6.	Group	$\boldsymbol{R} \subseteq R_e$	U `from` R' `to` $\boldsymbol{R}$	`if` $\boldsymbol{R}$ U `from` R'
7.	Group	$\boldsymbol{R} \subseteq R_e$	U `from` $\boldsymbol{R}$ `to` R'	`if` $\boldsymbol{R}$ U `to` R'
8.	Relative role	$\boldsymbol{R}[e] \subseteq R_e$	U `from` $R'[b]$ `to` $\boldsymbol{R}[e]$	`if` $\boldsymbol{R}$`[apply(`b`, `e`)]` U `from` R'`[inv(e)]`
9.	Relative role	$\boldsymbol{R}[b] \subseteq R_e$	U `from` $\boldsymbol{R}[b]$ `to` $R'[e]$	`if` $\boldsymbol{R}[b]$ U `to` $R'[e]$
10.	Choice sender	$\boldsymbol{R} = R_e$ or $\boldsymbol{R} \in R_e$	`choice at` $\boldsymbol{R}$ { G_1 } or ... or { G_N }	`choice at` $\boldsymbol{R}$ { L_1 } or ... or { L_N }
11.	Choice receiver		`choice at` R' { G_1} or ... or { G_N}	`choice at` R' { L_1} or ... or { L_N}
12.	Recursion		`rec` l { G }	`rec` l { L }
13.	Continue		`continue` l	`continue` l
14.	Foreach		`foreach` (b) { G }	`foreach` (b) { L }
15.	All reduce		`allreduce` $op_c(T)$	`allreduce` $op_c(T)$

L and L_i correspond to the projection of G and G_i onto $\boldsymbol{R}$

Table 2 Examples of `apply()` and `inv()`

Range (b)	Expr. (e)	`apply( b, e)`	`inv( e)`
`i:1..N`	`i+1`	`i:2..N+1`	`i-1`
`i:1..3`	`i*2`	`i:2,4,6`	`i/2`
`i:1..3`	`i`	`i:1..3`	`i`
`i:0..3`	`1<<i`	`i:1,2,4,8`	`log(i, 2)`
`i:1..3`	`i%2`	`i:1,0,1`	Invalid

only one argument: the algorithm can be extended easily to multiple arguments using the same methods). Rule 8 uses two auxiliary transformations of expressions, `apply` and `inv`. Table 2 lists their examples. `apply` takes two arguments, a range with binding variable (b) and an expression using the binding variable (e). The expression is *applied* to both ends of the range to transform the relative expression into a well-defined range. `inv` calculates the inverse of a given expression, for example, the inverse of `i+1` is `i-1` and the inverse of `i*2+1` is `(i-1)/2`. In cases when an inverse expression cannot be derived, such as `i%2`, the expression will be calculated by expanding to all values in the range and instantiating every value bound by its binding variable (e.g., `i`).

A concrete example is given as follows, to project the statement

```
U from W[i:1..3] to W[(i+1)%2];
```

the statement will be expanded to

```
U from W[1] to W[0];
U from W[2] to W[1];
U from W[3] to W[0];
```

before applying the projection rules. In order to perform the range expansion above, the beginning and the end of the range must be known at projection time. For this reason, the projection algorithm returns failure if a statement uses parameterised roles with such expressions and the range of the expressions is defined with arbitrary constants (see Sect. 2.4). Otherwise, the expressions might expand infinitely and not terminate. This is the only situation which projection may fail, given a well-formed global protocol. The condition $\boldsymbol{R}[b] \subseteq R_e$ of Rule 9 means the range of b is within the range of the endpoint role R_e. For example, `W[i:1..2]` $\subseteq$ `W[1..3]`.

If a projection role matches the choice role ($\boldsymbol{R}$ in **`choice at`** $\boldsymbol{R}$) (Rule 10), then it means a selection statement, whose action is selecting a branching by sending a label. The child or blocks ($L_1 \ldots L_N$) are recursively projected, whereas if a projection role does not match the choice role (Rule 11), then the choice statement represents a branch statement, which is the dual of the selection. For recursion (Rule 12), continue (Rule 13) and foreach (Rule 14) statements are just kept in the projected endpoint protocol.

2.6 Collective operations

In addition to point-to-point message-passing, collective operations can also be concisely represented by Pabble. End point message-passing statements are interpreted differently depending on the declarations (i.e., parameters) in the global type. Figures 3, 4, 5 and 6 list the four basic messaging patterns and the interpretations of their projections: point-to-point, scatter (distribution), gather (collection) and all-to-all (symmetric distribution and collection). As shown in the figures, the combination of projected local statements and the type (i.e., single participant or group role) of the local role being projected are unique and can identify the communication pattern in the global protocol.

2.7 Correctness and termination of the projection

The parameterised session theory which Pabble is based on [9] has shown that, in the general case, projection and type checking are undecidable. Our first challenge for Pabble's design is to ensure the termination of well-formed checking and projection, without sacrificing the expressiveness. The theorems and proofs can be found in this section.

Point-to-Point

Pabble role declarations: `role A[1..M], role B[1..N]`

$A_1 \longrightarrow B_1$
$A_2 \longrightarrow B_2$
$A_3 \longrightarrow B_3$

Pabble statement	Projection of A	Projection of B
`U from A to B;`	`U to B;`	`U from A;`
`U from A[i] to B[j];`	`if A[i] U to B[j];`	`if B[j] U from A[i];`
`U from A[i:1..N] to B[i+1];`	`if A[i:1..N] U to B[i+1];`	`if B[i:2..N+1] U from A[i-1];`

Fig. 3 Point-to-point communication and Pabble representation

Scatter pattern

Pabble role declarations: `role A, role B[1..N], group C`

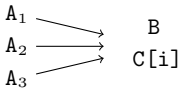
`A` → `B[i]` → C_1, C_2, C_3

Pabble statement	Projection of A/B	Projection of C
`U from A to C;`	`U to C;`	`if C U from A;`
`U from B[i] to C;`	`if B[i] U to C;`	`if C U from B[i];`

Fig. 4 Scatter pattern and Pabble representation

Pabble role declarations: `group A, role B, role C[1..N]`

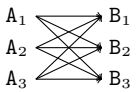
Gather pattern



Pabble statement	Projection of A	Projection of B/C
`U from A to B;`	`if A U to B;`	`U from A;`
`U from A to C[i];`	`if A U to C[i];`	`if C[i] U from A;`

Fig. 5 Gather pattern and Pabble representation

All-to-all pattern



Pabble role declarations: `group A, group B`

Pabble statement	Projection of A	Projection of B
`U from A to B;`	`if A U to B;`	`if B U from A;`
`U from All to All;`	`U to All; U from All;`	`U to All; U from All;`

Fig. 6 All-to-all pattern and Pabble representation

Theorem 1 (Termination) *Given global protocol G, the well-formed checking terminates; and given a well-formed global type G and an end point role R_e, projection G on R_e always terminates.*

Proof By the definition of the well-formedness conditions in Sect. 2.3, if a free variable appears in the range position, it is bound by either for-loops or the sender role in the interaction statement. In the case of the for-loop, we can apply the same reduction rules of the for-loop of the global types from Sect. 2 and apply the equality rules in [9, Fig. 15]. Hence, one can check, given R_e and $\boldsymbol{R}$, all of the conditions (in the second column) in Table 1 are decidable. For the projection, the only non-trivial projection rule is Rule 8. The termination of this rule is ensured by the termination of `apply(`b`,`e`)` and `inv(` e `)` . If `inv(`e`)` is not defined, we first check e has the finite range and use Rule 3 and 4 by expanding the interaction statements to all values in the range (as explained in Sect. 2.5). Hence, the projection algorithm always terminates. □

Note that the above theorem implies the termination of type checking (see Theorem 4.4 in [9]).

One of the benefits of using Pabble is that it provides the expressiveness required to be able represent collective interactions in MPI. The correctness of projections of these protocols is ensured by the projection rule of the groups in [7]. The special case of *U* **`from`** `all-`**`to`**`-all` follows the asynchronous subtyping rules in [18]. The correctness property which relates to ranges of Pabble follows:

Theorem 2 (Range) *The indices of roles appearing in a local protocol body do not exceed the lower and upper bounds stated in the global protocol ProtocolName(para) in* **`global protocol`** `ProtocolName(para){G}` *or the constant declarations (*`const` $N = n..m$*).*

Proof If the range relies on case (2), the correctness is ensured by linear programming. Other cases are straightforward since each condition in Table 1 checks whether roles conform to the bounds in the global protocol. □

3 Pabble examples

In Sect. 2.5, we describe how to obtain a local Pabble protocol by projection from a Pabble protocol. The local protocol can then be used as a blueprint to implement parallel programs. In this section, we run through two examples of local protocol projection, using a `Ring` protocol in Sect. 3.1 and a `MapReduce` protocol in Sect. 3.2, showing projection of protocols involving point-to-point and multicast collective applications, respectively.

Then, we present Pabble use cases in web services in Sect. 3.3 and remote procedure call (RPC) composition in Sect. 3.4, showing the capabilities of Pabble as a general-purpose parameterised protocol description language.

Finally, we show an implementation of a parallel linear equation solver Sect. 3.5 in MPI following a wrap-around mesh protocol designed in Pabble, demonstrating how Pabble can be used in practical programming. Additional Pabble examples from the Dwarfs [1], evaluation metric is available from our web page [21].

3.1 Projection example: Ring protocol

```
1  global protocol Ring(role Worker[1..N]) {
2    rec LOOP {
3      Data(int) from Worker[i:1..N-1] to
           Worker[i+1];
4      Data(int) from Worker[N] to Worker[1];
5      continue LOOP;
6    }
7  }
```

We now run through the projection of the `Ring` protocol in Sect. 1 as an example. Local protocols are generated from the global protocols. From the perspective of a projection tool, to write a protocol for an endpoint, we start with **`local protocol`** followed by the name of the protocol and the endpoint role it is projected for. Since the only role of the `Ring` protocol is `Worker` which is a parameterised role, we use the full definition of the parameterised role, `Worker[1..N]`. Then, we list the roles used in the proto-

col inside a pair of parentheses, similar to function arguments in a function definition in C. Note that if the projection role is in the list, we exclude it because the local protocol itself is in the perspective of that role; however, since parameterised roles can be used on multiple endpoint roles, we allow parameterised roles to appear in the list of roles in the protocol. The first line of the projected protocol is thus given as follows:

```
1  local protocol Ring at Worker[1..N](role
       Worker[1..N])
```

We then copy the recursion statement to the local protocol, which will be present in all projected protocols.

```
2    rec LOOP {
```

Next, we take the first interaction statement from `Ring` protocol and project it with respect to `Worker`, applying the rules listed in Table 1. As the first statement involves a parameterised destination role, we apply Rule 7 to extract the receive portion of the interaction statement. The `apply()` function is applied to `i:1..N-1` and the relative expression `i+1` to obtain `2..N` for the role condition. The `inv()` of relative expression `i+1` is `i-1`, which will form the index of the sender role.

```
3    if Worker[i:2..N] Data(int) from Worker[i
       -1] ;
```

Since `Worker` also matches the source parameterised role, Rule 8 is applied to get the send portion of the interaction statement.

```
4    if Worker[i:1..N-1] Data(int) to Worker[i
       +1];
```

Then, we move on to the second statement of the global protocol, `Data(int)` **`from`** `Worker[N]` **`to`** `Worker[1];`. Similar to the previous statement, we apply Rule 3 and Rule 4 to obtain the respective receive and send statements in the local protocol.

```
5    if Worker[1] Data(int) from Worker[N];
6    if Worker[N] Data(int) to Worker[1];
```

Finally, we apply Rule 13 to trivially copy the **`continue`** statement to the local protocol.

```
7    continue LOOP; }
```

The resulting local protocol is the following, as shown in Sect. 1.

```
1  local protocol Ring at Worker[1..N](role
       Worker[1..N]) {
2    rec LOOP {
3      if Worker[i:2..N]   Data(int) from
         Worker[i-1];
4      if Worker[i:1..N-1] Data(int) to Worker[
         i+1];
5      if Worker[1]        Data(int) from
         Worker[N];
6      if Worker[N]        Data(int) to Worker
         [1];
7      continue LOOP;
8    }
9  }
```

3.2 Projection example: MapReduce protocol

The following example shows another parameterised protocol, which represents the map–reduce pattern of work distribution and reduction. This example uses a common parallel programming idiom, collective operations. In contrast to the previous example, there are more than one declared role in the protocol, and one of the role is an ordinary nonparameterised role.

```
1  global protocol MapReduce(role Master, role
       Worker[1..N], group Workers={Worker
       [1..N]}){
2    Map(int) from Master to Workers;
3    Reduce(int) from Workers to Master;
4  }
```

Listing 1 MapReduce global protocol

In this protocol, the statements involve two roles, one of which is an ordinary role `Master` (in the sense that it is non-parameterised), and the other is a parameterised role `Worker[i:1..N]`. The `Worker` parameterised role represents a group of related roles, but do not expand to multiple explicit message-passing statements. We further declare a group role `Workers` which include all the `Worker` roles as members. The statement in Line 2 is a *scatter* operation by which the `Master` distributes a message of type `Map(int)` to each of the named endpoints in `Workers` group, `Worker[1]` to `Worker[N]`. The statement in Line 3 is a *gather* operation, the reverse of the scatter, which the `Master` role collects messages of type `Reduce(int)` from the members of the `Workers` group. Figure 7 depicts the interactions in the protocol.

Listing 2 shows the local protocol of MapReduce at the `Master` role. Since `Master` is a nonparametric participant, Rule 2 and 1 are applied to get Line 2 and 3, respectively. This results in a protocol body without conditional interactions.

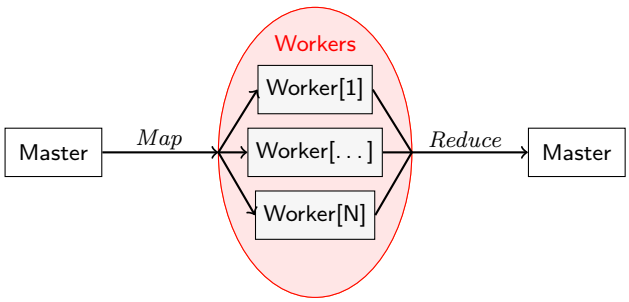


Fig. 7 Topology of the MapReduce protocol

```
1  local protocol MapReduce at Master(role
       Master, role Worker[1..N], group
       Workers={Worker[1..N]}) {
2    Map(int) to Workers;
3    Reduce(int) from Workers;
4  }
```

Listing 2 Master endpoint from MapReduce protocol

The local protocol of `Worker` for `MapReduce` is similarly derived by applying the projection rules. Since `Workers` is a group role and a subset of `Worker[1..N]`, Rule 6 and 7 are applied to get Line 2 and 3.

```
1  local protocol MapReduce at Worker[1..N](
       role Master, role Worker[1..N], group
       Workers={Worker[1..N]}) {
2    if Workers Map(int) from Master;
3    if Workers Reduce(int) to Master;
4  }
```

Listing 3 Worker endpoint from MapReduce protocol

3.3 Use case: web services

Pabble is inspired by applications in the domain of parallel programming, but the parametric nature of Pabble as a protocol language allows us to express interactions with more flexibility while keeping the protocols succinct.

Quote Request protocol specification (C-UC-002) is the most complex use case in [32] published by W3C Web Services Choreography Working Group [31].

```
1  global protocol WebService (role Buyer,
       role Supplier[1..S], role Manufacturer
       [1..M]) {
2    Quote() from Buyer to Supplier[1..S];
3    rec RENEGOTIATE_MANUFACTURER {
4      foreach (j:1..M) {
5        Item() from Supplier[i:1..S] to
             Manufacturer[j];
6        Quote() from Manufacturer[j] to
             Supplier[1..S];
7      }
8      // Gather
9      Quote() from Supplier[1..S] to Buyer;
10     foreach (i:1..S) { // (3)
11       rec RETRY_NEGOTIATION {
12         choice at Buyer {
13           // Buyer accepts quote and place
                 orders (4a)
14           ok() from Buyer to Supplier[i];
15         } or {
16           // Buyer modifies quotes and send
                 back to supplier (4b)
17           modify(Quote) from Buyer to
                 Supplier[i];
18           choice at Supplier[i] {
19             // Supplier agrees
20             // to modified quote (5a)
21             ok() from Supplier[i] to Buyer;
22           } or {
23             // Supplier modifies quote again
                   (5b)
24             retry(Quote) from Supplier[i] to
                   Buyer;
25             // Retry Supplier[i]-Buyer
                   negotiation
26             continue RETRY_NEGOTIATION;
27           } or {
28             // Reject (5c)
29             reject() from Supplier[i] to
                   Buyer;
30           } or {
31             // Supplier renegotiate with
                   Manufacturers for quote (5d)
32             renegotiate() from Supplier[i] to
                    Buyer;
33             continue RENEGOTIATE_MANUFACTURER
                   ;
34           }
35         }
36       } // Try NEXTSUPPLIER
37     }
38   } }
```

Listing 4 Web Services use case

```
1  local protocol WebService at Buyer (role
       Supplier[1..S], role Manufacturer[1..M
       ]) {
2    Quote() to Supplier[1..S];
3    rec RENEGOTIATE_MANUFACTURER {
4      Quote() from Supplier[1..S];
5      foreach (i:1..S) {
6        rec RETRY_NEGOTIATION {
7          choice at Buyer {
8            ok() to Supplier[i];
9          } or {
10           modify(quoteType) to Supplier[i];
11           choice at Supplier[i] {
12             ok() from Supplier[i];
13           } or {
14             retry(quoteType) from Supplier[i
                   ];
15             continue RETRY_NEGOTIATION;
16           } or {
17             reject() from Supplier[i];
18           } or {
19             renegotiate() from Supplier[i];
20             continue RENEGOTIATE_MANUFACTURER
                   ;
21           }
22         } // choice at Buyer
23       }
24     }
25   } }
```

Listing 5 Buyer endpoint of WebService

It describes the interaction between a buyer who interacts with multiple suppliers who in turn interact with multiple manufacturers in order to get a quote for some goods or services.

The basic steps of the interaction is as follows:

1. A buyer requests a quote from a set of suppliers
2. All suppliers forward the quote request of the items to their manufacturers

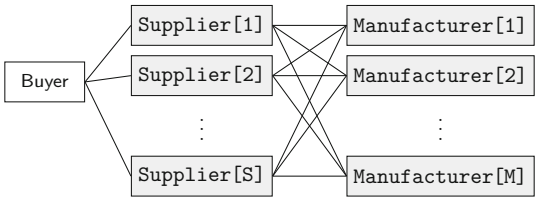


Fig. 8 Web Services Quote Request Interaction

3. The suppliers interact with their manufacturers to build the quotes for the buyer, which is then sent back to the buyer
4. (a) Either the buyer agrees with the quotes and place the orders
   (b) Or the buyer modify the quote and send back to the suppliers
5. In the case, the supplier received an updated quote request (4b)
   (a) Either the supplier respond to updated quote request by agreeing to it and sending a confirmation message back to buyer
   (b) Or the supplier respond to the update quote request by modifying it and sending back to buyer and the buyer goes back to step 4
   (c) Or the supplier respond to the update quote request by rejecting it
   (d) Or the supplier renegotiate with the manufacturers, in which case we return to step 3

Figure 8 shows the interactions between different components in the Quote Request use case. We set the generic number `S` for suppliers and `M` for manufacturers. The interactions are described as a Pabble global protocol in Listing 4. In the protocol, we omitted the implicit `requestIdType` from the payload type in all of the messages which keeps track of states of each role in the stateless web transport.

The `Buyer` initiates the quote request on Line 2, when it broadcasts a `Quote()` message to all `Suppliers`. Then, on Line 4–7 each of the `Supps` forward the quote requests to their respective `Manufacturers` and get a reply from each of them by a series of gather and scatter interactions. Next, the `Suppliers` reply to the `Buyer` on Line 9, and the `Buyer` then decides between accepting the offer straight away (Line 14, outcome 4a), or sending a modified quote request (Line 17, outcome 4b). If a `Supp` received a modified quote, it decides between accepting the modified quote (Line 21, outcome 5a), rejecting the modified quote straight away (Line 29, outcome 5c) or modifying the quote and renegotiating with `Buyer` (Line 24, outcome 5b). It is also possible that the `Supplier` renegotiates with its `Manufacturers` again, so it notifies the `Buyer` and returns back to the initial negotiation phase (Line 32, outcome 5d). The projected endpoint protocol for `Buyer` is Listing 5.

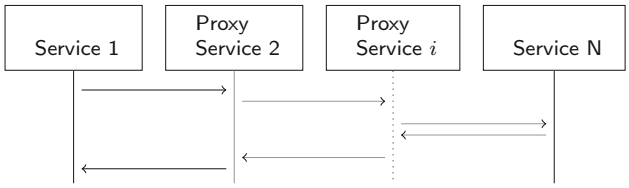


Fig. 9 RPC request/response chaining

3.4 Use case: RPC composition

We present a use case from the Ocean Observatories Initiative Project [28]. The use case describes a high-level Remote Procedural Call (RPC) request/response protocol between layers of proxy services. An application sends a request to a high-level service, and the service is expected to reply to the application with a result. If the service does not provide the requested service, then this high-level service will issue a request to a lower level service which can process the request. This request-response protocol is chained between services in each level until a low-level service is reached.

Figure 9 describes the chaining of RPC-style request/response protocol. A request is routed to the most relevant service provider through multiple proxy services hidden from higher level services. The request routes through a multi-hop path from the requester to the resources. The reply is routed in reverse through the same participant proxy services back to the requester.

We represent this series of interactions using a Pabble protocol outlined below. The participants, `Service[1..N]`, represents a proxy service in each of the levels. `Service[1]` is the requester and `Service[N]` is the actual service provider. A `Request()` message is sent from a `Service` to the `Service` in the level directly below, until it reached `Service[N]` which will process the request and reply to the higher level service with a `Response()`. Using a **`foreach`** loop with decrementing indices, the `Response()` is cascaded to the originating service, `Service[1]`. The Pabble protocol is shown in Listing 6.

```
1 global protocol RPCChaining(role Service
      [1..N]) {
2   foreach (i:1..N-1) {
3     Request() from Service[i] to Service[i
          +1];
4   }
5   // Request() processed by Service[N] to
        give Response()
6   foreach (i:N..2) {
7     Response() from Service[i] to Service[i
          -1];
8   }
9 }
```

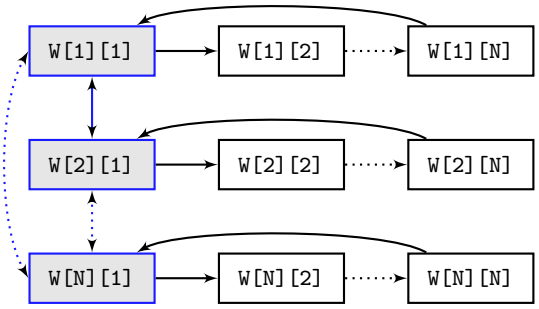


Fig. 10 N^2-node wraparound mesh topology

Listing 6 RPC request/response chaining

As the request and response phase are symmetric and involve the same participants, we are able to compact the multi-layer protocol to only using two **foreach** loops, each with one parameterised interaction statement. `N` can be an arbitrary constant to allow maximum flexibility in the protocol. This simple and concise representation of complex RPC chaining protocol is possible because of the index notation in Pabble.

3.5 Implementation example: Linear equation solver

Listing 9 shows an example implementation outline for a linear equation solver using a wraparound mesh, which follows the Pabble protocol in Listing 7. The topology is illustrated in Fig. 10. The example is given in message-passing interface (MPI), the standardized API for developing message-passing applications in parallel computing.

```
1  global protocol Solver(role W[1..N][1..N],
       group Col={W[1..N][1]}) {
2    rec CONVERGE {
3      Ring(double) from W[i:1..N][j:1..N-1] to
           W[i][j+1];
4      Ring(double) from W[i:1..N][N] to W[i
           ][1];
5
6      // Vertical propagation - Group-to-Group
7      (double) from Col to Col;
8      continue CONVERGE;
9    }
10 }
```

Listing 7 Linear equation solver protocol

The protocol above describes a wraparound mesh that performs a ring propagation between `W` (for worker) in the same row (Line 3–4), and the result of each `W` row is distributed to all `W`s in the first column (i.e., `W[*][1]`) using a group-to-group distribution on Line 7. The global protocol is then automatically projected into its local protocol shown in Listing 8 below. Developers can then implement the application using its local protocol as a guide.

```
1  local protocol Solver at W(role W[1..N][1..
       N], group Col={ W[1..N][1] }) {
2    rec CONVERGE {
3      if W[i:1..N][j:2..N] Ring(double) from W
           [i][j-1];
4      if W[i:1..N][j:1..N-1] Ring(double) to W
           [i][j+1];
5      if W[i:1..N][1] Ring(double) from W[i][N
           ];
6      if W[i:1..N][N] Ring(double) to W[i][1];
7
8      // Vertical propagation - Group-to-Group
9      if Col (double) from Col;
10     if Col (double) to Col;
11     continue CONVERGE;
12   }
13 }
```

Listing 8 Linear equation solver local protocol

Note the similarity of the local protocol and the structure of the MPI implementation in Listing 9. In particular, the conditional send and receive in MPI can directly correspond to the role conditions in the local protocol which was derived from the global protocol by projection.

```
1  MPI_Init(&argc, &argv); // Start of
       protocol
2  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //
       Process ID
3  MPI_Comm_size(MPI_COMM_WORLD, &size); // #
       of Process
4  MPI_Comm Col; int N = (int)sqrt(size);
5  ...
6  /* Calculate condition for W[i:1..N][j:2..N
       ] */
7  if (2 <= rank
8   MPI_Recv(buf, cnt, MPI_DOUBLE, rank-1/*W[i
       ][j-1]*/, Ring, MPI_COMM_WORLD);
9  /* Calculate condition for W[i:1..N][j:1..N
       -1] */
10 if (1 <= rank
11  MPI_Send(buf, cnt, MPI_DOUBLE, rank+1/*W[i
       ][j+1]*/, Ring, MPI_COMM_WORLD);
12 /* Calculate condition for W[i:2..N][j:1..N
       ] */
13 if (2 <= rank/N+1 && rank/N+1 <= N)
14  MPI_Send(buf, cnt, MPI_DOUBLE, rank-N*1/*W
       [i-1][j]*/, Ring, MPI_COMM_WORLD);
15 /* Calculate condition for W[i:1..N-1][j
       :1..N] */
16 if (1 <= rank/N+1 && rank/N+1 <= N-1)
17  MPI_Send(buf, cnt, MPI_DOUBLE, rank+N*1/*W
       [i+1][j]*/, Ring, MPI_COMM_WORLD);
18
19 /* Distribute vertically: Group-to-Group on
       'Col' group communicator */
20 if (rank
21  MPI_Allgather(buf_col, cnt_col, MPI_DOUBLE
       ,
22                buf_col, cnt_col, MPI_DOUBLE
                    , Col);
23 ...
24 MPI_Finalize(); // End of protocol
```

Listing 9 MPI implementation for Solver protocol

4 Type checking

Given the local protocol and the implementation, we propose a session type checker to verify the conformance of the implementation against the projected local protocol. Conformance of end point programs against the projected protocol will yield communication-safe parallel programs.

Pabble local protocols have similar structure to that of MPI programs. Both Pabble protocols and MPI programs are designed such that a single source code representing multiple end points, a result of the single-program multiple-data (SPMD) parallel programming model. The core communication primitives of MPI can correspond to Pabble statements, as demonstrated in Listing 9. In addition, collective operations such as broadcast (**`MPI_Bcast`**) or all-reduce (**`MPI_Allreduce`**) can be supported by the collective operation correspondence in Sect. 2.5.

Challenges for a complete MPI type checker In [20], Ng et al. introduced a session type checker for a nonparameterised protocol language and a simple session programming API. We face a number of challenges when building a complete type checker using the same methodology for Pabble, which is a dependent protocol language and MPI, which is a standard parameterised implementation API. The Pabble language with its well-formedness checks reduces the undecidability issues in the role representation by using integer instead of general indices. The type checking process will compare the protocol against a simplified, canonical local protocol extracted from the implementation, which still posts a challenge in the process of protocol extraction. In particular, inferring source and destination processes from parametric source code is non-trivial. MPI uses process IDs (or ranks) to identify processes, and it is valid to perform numeric operations on the ranks to efficiently calculate target processes. This allows ways of exploiting the C language features while remaining a valid program. For example, instead of using a conventional conditional statement, an MPI function call of this form may be used:

```
MPI_Send(buf, cnt, MPI_INT, rank%2? rank+1:
    rank-1, ...)
```

where the process ID, `rank`, is being used as a boolean, thus a straightforward analysis of `rank` usages would not be sufficient. In order to correctly calculate target processes of the interactions, it will be necessary to simulate rank calculations by techniques such as symbolic execution or combinations of runtime techniques.

5 Conclusion

This article introduced a new global protocol description language, Pabble, and applied it to ensure deadlock-free and type-safe communications in parallel programs. Local protocols projected from a parameterised global protocol and we outlined a methodology to specify and type-check MPI parallel programs for safe parallel programs. Our global protocols and local protocols bring the expressiveness of Scribble to new levels, overcoming the issue of the underlying parameterised multiparty session type theory [9] by a careful design choice for indices based on [16]. Combining with the multirole theory from [7], Pabble can represent and type-check representative MPI collective operators. We are not aware of any prior framework which is uniformly applicable to a safety guarantee for message-passing parallel programs which run over complex topologies, through static, low-cost type checking as compared to model checking.

Through our examples presented in this article, we have showed that the Pabble language is not limited to high-performance parallel applications. The examples, including web services and RPC, cover a broad category of interaction-centric scalable distributed applications. Our simple, formally based language provides an approach for designing services and applications with safe interaction patterns.

6 Related work

6.1 Formal verification for parallel applications

Formal verification for message-passing parallel programming has been actively studied in the area of MPI parallel applications. A recent survey [10] summarizes a wide range of model checking-based verification methods for MPI. Among them, ISP [29] is a dynamic verifier which applies model-checking techniques to identify potential communication deadlocks in MPI. Their tool uses a fixed test harness and in order to reduce the state space of possible thread interleavings of an execution, the tool exploits an independence between thread actions. Later in [30], the authors improved its scheduling policy to gain efficiency of the verification. While their approach aims to cover common deadlock patterns in MPI programs, it is still limited to a finite number of tests. Our approach does not rely on external testing, and all session typable programs are guaranteed communication-safe and deadlock-free by a low-cost static code generation and type checking.

TASS [26] is another tool that combines symbolic execution [25] and model-checking techniques to verify safety properties of MPI programs. The tool takes a C/MPI application and an input $n \geq 1$ which restricts the input space, then constructs an abstract model with n processes and checks its functional equivalence and deadlocks by executing the model of the application. TASS does not verify properties for an unbounded number of communication participants nor treat parameterisation, whereas we can work with

message-passing programs where the number of participants is unknown at compile time, if they are written in well-formed, projectable Pabble.

6.2 Formally based MPI languages

Pilot [5] is a parallel programming library built on standard MPI to provide a simplified parallel programming abstraction based upon CSP. The communication is synchronous and channels are untyped to facilitate reuse for different types. The implementation includes an analyser to detect communication deadlock at runtime. Our proposed typechecker is static and is able to detect and prevent deadlocks before execution.

Interprocedural control flow graph (ICFG) [27] and parallel control flow graph (pCFG) [4] are techniques to analyze MPI parallel programs for potential message leak errors. Their approach extends a traditional data-flow analysis by connecting control flow graphs of concurrent processes to their communication edges in order to derive the communication pattern and topology of a parallel program. They take a bottom-up engineering-based approach, in contrast to our formally based, top-down global protocol approach, which can give a high-level understanding of the overall communication by design, in addition to the communication safety assurance by multiparty session types.

6.3 Parameterised multiparty session types

Previous work from Ng et al. [20] introduces a C programming framework based on multiparty session types (MPSTs), but it does not treat parameterisation. Hence, the user needs to explicitly describe all interactions in the protocol, and the type checker does not work if the number of participants is unknown at compile time. Pabble's theoretical basis is developed in [9] where parameterised MPSTs are formalized using the dependent type theory of Gödel's System $\mathcal{T}$. The main aim in [9] is to investigate the decidability and expressiveness of parameterisations of participants. Type checking in [9] is undecidable when the indices are not limited to decidable arithmetic subsets or the number of the loop in the parameterised types is infinite. The design of Pabble is inspired by the LTSA tool from a concurrency modeling text book used for the undergraduate teaching in the authors' university over two decades [16]. The notations for parameterisations from the LTSA tool offers not only practical restrictions to cope with the undecidability of parameterised MPSTs [9], but also concise representations for parameterised parallel languages. Our work is the first to apply parameterised MPSTs in a practical environment and one foremost aim of our framework with Pabble and parameterised notation is to be developer-friendly [24] without compromising the strong formal basis of session types.

6.4 Dependent typing systems

Liquid Type [23] is a dependent typing system to automatically infer memory safety properties from program source code without using verbose annotations. The work [22] introduced an analyser for the C language in the low-level imperative environment based on Liquid Types and refinement types. The recent work on Liquid Types [15] applied the tool with SMT solvers to assist parallelisation of code regions by determining statically whether parallel threads will run on disjoint shared memory without races. Our work applies dependent session types to guarantee different kinds of safety, communication safety and deadlock freedom, in explicit message-passing based distributed and parallel programming rather than shared memory concurrency. It is an interesting future topic to integrate with model-checking tools to handle projectability with more complex indices in addition to functional correctness of session programs.

6.5 Session-based approaches to parallel programming

A recent work [11,17] aims to use session types for deductive verification of MPI programs. A new type language is designed specifically for MPI and they used VCC, a concurrent C verifier tool to verify correctness of MPI against the type language. While the Pabble language was designed with influences from parallel programming APIs and parallel programming use cases, the language was designed to be an independent high-level abstraction over distributed interactions. As a result, our language makes no assumption about the execution environment (e.g., collective loops in MPI), and allows Pabble to represent general protocols from distributed systems or web services with distinct roles as shown in the examples.

7 Future work

Future works include extending Pabble and the underlying theory with support for modeling process creation and destroy, such as dynamic multirole approach described in [7].

A number of enhancements are planned for Pabble including support for annotations which can complement the protocol description to specify assertions. The type checking process can use the extra constraints or conditions provided to combine with model checkers to also assure functional correctness of the overall application. Annotations will also enable integration with runtime monitoring described in [14] for a combined static and dynamic approach to communication correct application using Pabble.

An approach to generate distributed parallel application is in the works, using a combination of Pabble protocol, which

describes the interaction aspects of the application, and computation code, which describes the sequential computation behavior of the application.

Acknowledgments The work is funded by EPSRC EP/K034413/1, EP/K011715/1 and EP/L00058X/1, EU project FP7-612985 (UpScale) 257906, 287804 and 318521.

References

1. Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67
2. Aspinall D, Hofmann M (2005) Advanced topics in types and programming languages, chap. Dependent types. MIT Press, Cambridge
3. Bettini L, Coppo M, D'Antoni L, De Luca M, Dezani-Ciancaglini M, Yoshida N (2008) Global progress in dynamically interleaved multiparty sessions. In: CONCUR 2008, LNCS, vol 5201, Springer, Berlin, pp 418–433
4. Bronevetsky G (2009) Communication-sensitive static dataflow for parallel message passing applications. In: CGO'09, IEEE, pp 1–12
5. Carter J, Gardner WB, Grewal G (2010) The Pilot approach to cluster programming in C. In: IPDPSW, IEEE, pp 1–8
6. Castagna G, Dezani-Ciancaglini M, Padovani L (2012) On global types and multi-party session. LMCS 8(1)
7. Deniélou PM, Yoshida N (2011) Dynamic multirole session types. In: POPL, ACM, pp 435–446
8. Deniélou PM, Yoshida N (2012) Multiparty session types meet communicating automata. In: ESOP, LNCS, vol 7211, Springer, Berlin, pp 194–213
9. Deniélou PM, Yoshida N, Bejleri A, Hu R (2012) Parameterised multiparty session types. LMCS 8(4)
10. Gopalakrishnan G et al (2011) Formal analysis of MPI-based parallel programs. Commun ACM 54(12):82–91
11. Honda K, Marques E, Martins F, Ng N, Vasconcelos V, Yoshida N (2012) Verification of MPI programs using session types. In: EuroMPI'12, LNCS, vol 7490
12. Honda K, Mukhamedov A, Brown G, Chen TC, Yoshida N (2011) Scribbling interactions with a formal foundation. In: ICDCIT, LNCS, vol 6536, Springer, Berlin, pp 55–75
13. Honda K, Yoshida N, Carbone M (2008) Multiparty asynchronous session types. In: POPL'08, pp 273–284
14. Hu R, Neykova R, Yoshida N, Demangeon R (2013) Practical interruptible conversations: Distributed dynamic verication with session types and python. In: RV 2013, LNCS, vol 8174, pp 148–130
15. Kawaguchi M, Rondon P, Bakst A, Jhala R (2012) Deterministic parallelism via liquid effects. In: PLDI'12, pp 45–54
16. Magee J, Kramer J (2006) Concurrency: state models and Java programs, 2nd edn. Wiley, New York
17. Marques ERB, Martins F, Vasconcelos VT, Ng N, Martins ND (2013) Towards deductive verification of mpi programs against session types. In: PLACES'13, EPTCS, vol 137, pp 103–113
18. Mostrous D, Yoshida N, Honda K (2009) Global principal typing in partially commutative asynchronous sessions. In: ESOP, LNCS, vol 5502, pp 316–332
19. Ng N, Yoshida N (2014) Pabble: parameterised scribble for parallel programming. In: PDP 2014, IEEE (to appear)
20. Ng N, Yoshida N, Honda K (2012) Multiparty session C: safe parallel programming with message optimisation. In: TOOLS, LNCS, vol 7304, Springer, Berlin, pp 202–218
21. Pabble project page. http://www.doc.ic.ac.uk/~cn06/pabble
22. Rondon PM, Kawaguchi M, Jhala R (2010) Low-level liquid types. In: POPL'10, pp 131–144. http://dl.acm.org/citation.cfm?id=1375602
23. Rondon PM, Kawaguci M, Jhala R (2008) Liquid types. In: PLDI'08, pp 159–169
24. Scribble homepage. http://scribble.github.io
25. Siegel S, Mironova A, Avrunin G, Clarke L (2008) Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM 17(2):1–34
26. Siegel SF, Zirkel TK (2011) Automatic formal verification of MPI-based parallel programs. In: PPoPP'11, ACM Press, p 309
27. Strout M, Kreaseck B, Hovland P (2006) Data-flow analysis for MPI programs. In: ICPP'06, IEEE, pp 175–184
28. Use cases from OOI project. https://confluence.oceanobservatories.org/display/CIDev/Identify+required+Scribble+extensions+for+advanced+scenarios+of+R3+COI
29. Vo A, Vakkalanka S, DeLisi M, Gopalakrishnan G, Kirby RM, Thakur R (2009) Formal verification of practical MPI programs. In: PPoPP'09, pp 261–270
30. Vo A et al (2010) A scalable and distributed dynamic formal verifier for MPI programs. In: SC'10, IEEE, pp 1–10
31. W3C Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/
32. Web Services Choreography Requirements. http://www.w3.org/TR/ws-chor-reqs/
33. Xi H, Pfenning F (1998) Eliminating array bound checking through dependent types. In: PLDI '98, pp 249–257