Pabble: Parameterised Scribble for Parallel Programming

Nicholas Ng
Imperial College London, UK
Email: nickng@doc.ic.ac.uk

Nobuko Yoshida
Imperial College London, UK
Email: n.yoshida@imperial.ac.uk

***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 paper 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 endpoint projection and type checking.**

I. 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 [1].

Our proposal for safety assurance for message-passing programs is based on *multiparty session types* [2]. 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 [3], which the authors proposed a programming framework for message-passing parallel algorithms, centring 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 [4], [5] 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,
2    role Worker3, role Worker4) {
3    rec LOOP {
4      Data(int) from Worker1 to Worker2;
5      Data(int) from Worker2 to Worker3;
6      Data(int) from Worker3 to Worker4;
7      Data(int) from Worker4 to Worker1;
8      continue LOOP; }}
```

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; }}
```

`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

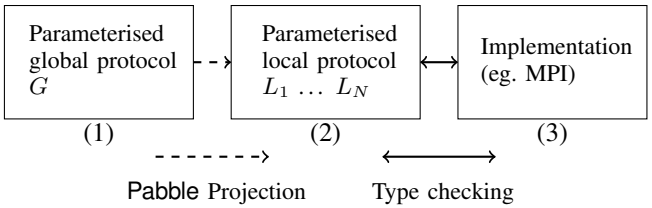

Figure 1: Pabble parallel programming workflow.

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

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 endpoints 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 [6], [7], the underlying parameterised session type theory [8] 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 modelling concurrent Java [9]. For example, notations `Worker[i:1..N-1]` and `Worker[j+i]` in the Ring protocol are from [9]. 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.

Overview: Figure 1 shows the relationships between the three layers: global protocols, local protocols and implementation. (1) A programmer first designs a global protocol using Pabble. (2) Then our Pabble tool automatically project 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 message-passing based interaction as global and local protocols in (1) and (2), and outline how a session type checker for MPI (3) can be implemented with Pabble. The contributions of this paper are:

- The first design and implementation of parameterised session types in a global protocol language (Pabble) (§ II-A). The protocols can represent complex topologies with arbitrary number of participants, enhancing expressiveness and modularity for practical message-passing parallel programs.
- The projection algorithm (§ II-D) for Pabble to check the well-formedness of parameterised global protocols (§ II-B) and to generate parameterised local protocols from well-formed parameterised global protocols (§ II-C). A correctness and termination proof of the projection algorithm is also presented (§ II-E).
- A complete use case of Pabble describing a parallel linear equation solver (§ III-B), and a methodology for type checking (§ III-C) written in MPI using Pabble.

Additional use cases of Pabble in different settings include covering common parallel topologies described in Dwarf [10], Web Service Choreography Language Specification [11] and the Ocean Observatories Initiative [12]. All program examples, additional use cases and implementations are available from [13].

II. Pabble: Parameterised Scribble

Scribble [5] is a developer friendly notation for specifying application-level protocols based on the theory of multiparty session types [2], [14]. This section introduces an evolution of Scribble with parameterised multiparty session types (Pabble), defines its endpoint projection and proves its correctness.

A. Syntax of Pabble

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 h, and the number of arguments corresponds to the dimension of the array of roles: for example, `Worker[1..4][1..2]` denotes a 2-D 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,

Global Pabble

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

Parameter

para ::= `role` R_d, ..., `group` *str*={R_d, ...}, ...

Global protocol body

G ::= $l(T)$ `from` R `to` R;
| `choice at` R { G } `or` { G }
| `foreach` (b) { G } | `allreduce` $op_c(T)$;
| `rec` l { G } | `continue` l; | G G

Payload type

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

Expression

e ::= e op e | *num* | $i, j, k, ...$ | N
op ::= op_c | - | / | % | << | >> | `log` | ...
op_c ::= + | * | ...

Role

R_d ::= *str* | *str*[e..e]...[e..e]
R ::= *str* | *str*[h]...[h] | `All`
h ::= b | e b ::= i : e..e

Local Pabble

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

Local protocol body

L ::= [`if` R] $l(T)$ `from` R; | [`if` R] $l(T)$ `to` R;
| `choice at` R { L } `or` { L }
| `foreach` (b) { L } | `allreduce` $op_c(T)$;
| `rec` l { L } | `continue` l; | L L

Figure 2: Pabble Syntax.

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

creates a group which consists of two `Worker`s. 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 logical 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 [9], 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 `inf` respectively, i.e. `const N = 0..inf`, where `inf` is a special value representing infinity. Binding range expression r 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 any 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,...,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 of 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.

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` $R_e(...)${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. 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.

B. 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 $\mathsf{fv}(i : e_1...e_n) = \cup\mathsf{fv}(e_j)$ and $\mathsf{fv}(\mathtt{foreach}(b)\{G\}) = (\mathsf{fv}(b) \cup \mathsf{fv}(G)) \setminus \mathsf{bv}(b)$ and $\mathsf{bv}(i : e_1...e_n) = \{i\}$. Others are inductively defined.

1) In a global protocol role declaration, `global protocol`, indices outside of declared range are in-

valid, 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 $R_1 = Role_1[h_1]..[h_n]$ and $R_2 = Role_2[e'_1]..[e'_m]$ (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 $l(T)$ **from** `Worker[i:1..N-1][j:1..N]` **to** `Worker[i+1]` is invalid. Condition 2(a)(2) ensures variables used by the sender are declared by the 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, $l(T)$ **from** `Worker[i:1..N-1][j:1..N]` **to** `Worker[i+1][j]` is valid, while $l(T)$ **from** `Worker[i:1..N-1][j:1..N]` **to** `Worker[j][i+1]` is not. 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 [15], [16].

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

The basic constraints from the constants are $1 \leq$ `M`, `M` ≤ 3, $2 \leq$ `N` and `N` ≤ 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 an objective function to be $1 \leq 1$ & `M` $\leq$ `N` and $1 \leq 2$ & `M` $+1 \leq$ `N`, which are lower and upper bound inequalities of the two ranges. From them, we obtain `M` $+1 \leq$ `N` as a result. By comparing this against the basic constraints on the constants, we can check not all outcomes belong to the regions and thus this is not a communication-safe protocol (an example of a unsafe case is `M` $= 3$ and `N` $= 2$). On the other hand, if we alter Line 4 to `T from R[i:1..N-1] to R[i+1]`, the objective function is 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 a general case of (2) and is equivalent to **const** `N = 0..inf` where `inf` is a keyword to represent positive infinity.

In order to check that role indices are valid with infinite ranges, we enforce two simple restrictions. First, only one constant can be defined with `inf` in one global protocol. Secondly, when the index is infinite, 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..inf;
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]; }
```

if `N` is 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..inf;
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 [13], most of representative topologies with the arbitrary number of participants can be represented under these conditions.

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

Table II: Examples of `apply()` and `inv()`.

D. Endpoint projection

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

$$\texttt{global protocol}\ \textit{name(param)}\ \{\ G\ \}$$

is projected onto

$$\texttt{local protocol}\ \textit{name}\ \texttt{at}\ R_e\textit{(param)}\ \{\ L\ \}$$

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

Table I shows the projection of the body of global protocol G onto $\boldsymbol{R}$ at endpoint role R_e.

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.

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 only one argument: the algorithm be extended easily to multiple arguments using the same methods). Rule 8 uses two auxiliary transformations of expressions, `apply` and `inv`. Table II 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 the projection of `U from W[i:1..3] to W[(i+1)%2]`, which 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 § II-C). 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.

Collective operations: In addition to point-to-point message-passing, collective operations can also be concisely represented by Pabble. Endpoint message-passing statements are interpreted differently depending on the declarations (i.e. parameters) in the global type. Table III lists the possibilities of a projection and their respective meanings on the first column with respect the declarations in the global types on the second column. 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. We consider three possible kinds of roles: single participant (e.g. `A` or `A[1]`), group role (e.g. `A[1..N]` or as declared by (`group G`) and relative role (e.g. `A[i+1]`).

E. Correctness and termination of the projection

The parameterised session theory which Pabble is based on [8] 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

$\boldsymbol{R}$	Conditions	Global protocol	Local protocol projected onto $\boldsymbol{R}$ at R_e
1. Non-parametric participant	$\boldsymbol{R} = R_e$	U **from** R' **to** $\boldsymbol{R}$	U **from** R'
2. Non-parametric participant	$\boldsymbol{R} = R_e$	U **from** $\boldsymbol{R}$ **to** R'	U **to** R'
3. Parametric participant	$\boldsymbol{R} \in R_e$	U **from** R' **to** $\boldsymbol{R}$	**if** $\boldsymbol{R}$ U **from** R'
4. Parametric participant	$\boldsymbol{R} \in R_e$	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 at single participant	$\boldsymbol{R} = R_e$ or $\boldsymbol{R} \in R_e$	**choice at** $\boldsymbol{R}$ $\{ G_1 \}$ **or** $\ldots \{ G_N \}$	**choice at** $\boldsymbol{R}$ $\{ L_1 \}$ **or** $\ldots \{ L_N \}$
11. Choice		**choice at** R' $\{ G_1 \}$ **or** $\ldots \{ G_N \}$	**choice at** R' $\{ L_1 \}$ **or** $\ldots \{ L_N \}$
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** $op_c(T)$

Table I: Projection of G onto $\boldsymbol{R}$ at the end-point role R_e.
L and L_i correspond to the projection of G and G_i onto $\boldsymbol{R}$.

	Declaration	Pabble statement	Projection of A	Projection of B
Point-to-point	`role A, role B`	`U from A to B`	`U to B`	`U from A`
(Param. A/B)	`role A[1..N], B[1..M]`	`U from A[i] to B[j]`	`if A[i] U to B[j]`	`if B[j] U from A[i]`
(Parallel P2P)	`role A[1..N], B[1..M]`	`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]`
Gather	`group A, role B`	`U from A to B`	`if A U to B`	`U from A`
(Param. B)	`group A, role B[1..N]`	`U from A to B[i]`	`if A U to B[i]`	`if B[i] U from A`
Scatter	`role A, group B`	`U from A to B`	`U to B`	`if B U from A`
(Param. A)	`role A[1..N], group B`	`U from A[i] to B`	`if A[i] U to B`	`if B U from A[i]`
Group-to-group	`group A, B`	`U from A to B`	`if A U to B`	`if B U from A`
All-to-all		`U from All to All`	`U to All; U from All`	`U to All; U from All`

Table III: Meanings and projections of different forms of interaction statements.

checking and projection, without sacrificing the expressiveness. The proofs of the following theorems can be found in [13].

Theorem 1 (termination): Given global protocol G, the well-formed checking terminates; and given a well-formed global type G and an endpoint role R_e, projection G on R_e always terminates.

Note that the above theorem implies the termination of type checking (see Theorem 4.4 in [8]).

One of benefits of using Pabble is that it provides the expressiveness required to be able represent collective protocols of MPI. The correctness of projections of these protocols is ensured by the projection rule of the groups in [17]. 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` = $n..m$).

III. Examples and Type Checking

In § II-D, a local Pabble protocol is obtained by projecting from a Pabble protocol. The local protocol can then be used as a blueprint to implement parallel programs. In this section we run through an example of local protocol projection, followed by an implementation of a parallel linear equation solver following a wraparound mesh protocol.

A. Projection example: Ring protocol

We now run through the projection of the `Ring` protocol in § I 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 protocol 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 I. 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, which is `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 shown in § I.

B. Implementation example: Linear equation solver

Listing 2 shows an example implementation outline for a linear equation solver using a wraparound mesh, which follows the Pabble protocol in Listing 1. The topology is illustrated in Figure 3. The example is given in Message-Passing Interface (MPI), the most commonly used library for 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
7    (double) from Col to Col;
8    continue CONVERGE; }}
```

Listing 1: Linear equation solver protocol.

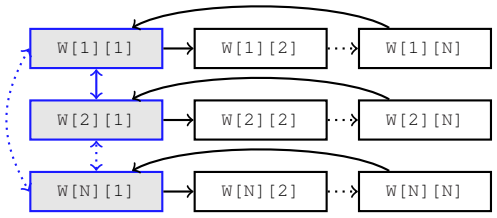

Figure 3: N^2-node wraparound mesh topology.

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]`). The local Pabble protocol of `Solver` is available in [13].

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

Listing 2: MPI implementation for Solver protocol

C. 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 endpoint 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 endpoints, 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 2. In addition, collective operations such as broadcast (**`MPI_Bcast`**) or all-reduce (**`MPI_Allreduce`**) are supported. The details are listed in [13].

Challenges for a complete MPI type checker: In [3], Ng et al. introduced a session type checker for a non-parameterised 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 some of the undecidability issues in the protocol representation. 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 more conventional conditional statements, **`MPI_Send`**`(buf, cnt, MPI_INT, rank%2 ? rank+1: rank-1, ...)` may be used and

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.

IV. Related and Future work

Formal verification and languages for parallel applications: Pilot [19] 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) [20] and parallel control flow graph (pCFG) [21] are techniques to analyse MPI parallel programs for potential message leak errors. 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.

Parameterised multiparty session types: Previous work from Ng et al. [3] 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 [8] where parameterised MPSTs are formalised using the dependent type theory of Gödel's System $\mathcal{T}$. The main aim in [8] is to investigate the decidability and expressiveness of parameterisations of participants. Type checking in [8] 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 modelling text book used for the undergraduate teaching in the authors' university over two decades [9]. The notations for parameterisations from the LTSA tool offers not only practical restrictions to cope with the undecidability of parameterised MPSTs [8], 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 [5] without compromising the strong formal basis of session types.

Future work: Future works include extending Pabble and the underlying theory with support for modelling process creation and destroy, such as dynamic multirole approach described in [17].

Acknowledgements.: The research leading to these results has received funding from EPSRC EP/F003757/01, EP/G015635/01 and the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521. The support by UK EPSRC, the HiPEAC NoE, the Maxeler University Program, and Xilinx is gratefully acknowledged.

References

[1] G. Gopalakrishnan *et al.*, "Formal analysis of MPI-based parallel programs," *Commun. ACM*, vol. 54, no. 12, pp. 82–91, 2011.

[2] K. Honda, N. Yoshida, and M. Carbone, "Multiparty asynchronous session types," in *POPL'08*, 2008, pp. 273–284.

[3] N. Ng, N. Yoshida, and K. Honda, "Multiparty Session C: Safe Parallel Programming with Message Optimisation," in *TOOLS*, ser. LNCS, vol. 7304. Springer, 2012, pp. 202–218.

[4] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida, "Scribbling interactions with a formal foundation," in *ICDCIT*, ser. LNCS, vol. 6536. Springer, 2011, pp. 55–75.

[5] "Scribble homepage," http://www.jboss.org/scribble.

[6] D. Aspinall and M. Hofmann, *Advanced Topics in Types and Programming Languages*. MIT, 2005, ch. Dependent Types.

[7] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *PLDI '98*, 1998, pp. 249–257.

[8] P.-M. Deniélou, N. Yoshida, A. Bejleri, and R. Hu, "Parameterised multiparty session types," *LMCS*, vol. 8, no. 4, 2012.

[9] J. Magee and J. Kramer, *Concurrency: state models and Java programs (2. ed.)*. Wiley, 2006.

[10] K. Asanovic *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[11] "W3C Web Services Choreography," http://www.w3.org/2002/ws/chor/.

[12] "Ocean observatories initiative," http://www.oceanobservatories.org/.

[13] "Full version." [Online]. Available: http://www.doc.ic.ac.uk/~cn06/pabble

[14] Bettini et al., "Global Progress in Dynamically Interleaved Multiparty Sessions," in *CONCUR 2008*, ser. LNCS, vol. 5201. Springer, 2008, pp. 418–433.

[15] P.-M. Deniélou and N. Yoshida, "Multiparty session types meet communicating automata," in *ESOP*, ser. LNCS, vol. 7211. Springer, 2012, pp. 194–213.

[16] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, "On global types and multi-party session," *LMCS*, vol. 8, no. 1, 2012.

[17] P.-M. Deniélou and N. Yoshida, "Dynamic multirole session types," in *POPL*. ACM, 2011, pp. 435–446.

[18] D. Mostrous, N. Yoshida, and K. Honda, "Global principal typing in partially commutative asynchronous sessions," in *ESOP*, ser. LNCS, vol. 5502, 2009, pp. 316–332.

[19] J. Carter, W. B. Gardner, and G. Grewal, "The Pilot approach to cluster programming in C," in *IPDPSW*. IEEE, 2010, pp. 1–8.

[20] M. Strout, B. Kreaseck, and P. Hovland, "Data-Flow Analysis for MPI Programs," in *ICPP'06*. IEEE, 2006, pp. 175–184.

[21] G. Bronevetsky, "Communication-Sensitive Static Dataflow for Parallel Message Passing Applications," in *CGO '09*. IEEE, 2009, pp. 1–12.