Deadlock-Free Parallel Regions for Projected Workflows

Benedikt Bollig
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF
Gif-sur-Yvette, France

Projected workflow languages describe a distributed protocol once and compile it to one local program per participant. Many agent workflows are not purely sequential: independent streams of tasks may proceed at the same time while sharing some participants, such as a user or a calendar service. The ZIPPERGEN workflow language gives deadlock-freedom guarantees for sequential coordination, in particular for LLM-agent workflows, but its core language has no parallel construct that supports shared agent lifelines. We add such a construct: a structured parallel-region operator whose branches are full workflows and may share lifelines. Projection assigns separate branch-local channel names to messages in different branches, although no channel annotations are written in the source program. A shared lifeline is projected to one local program with a local interleaving operator over the corresponding branch projections. The semantics of a parallel region keeps exactly the globally consistent shuffles of branch executions, in which messages are matched and causal order has no cycles. We prove that projection preserves the workflow semantics and yields deadlock-free distributed programs.

1 Introduction

Sequential composition in a projected workflow orders the actions of each participant that appears in both components. This order is often useful, but it is not always part of the intended protocol. For example, a command center may handle email and chat in parallel while both streams use the same user or calendar service. Forcing all email steps to precede all chat steps, or conversely, adds an unnecessary order. Removing that order still leaves constraints: each receive must have a matching send, and the local orders imposed by shared participants must not form a cycle.

ZIPPERGEN was introduced as a basic projected workflow language for multi-agent coordination [3]. A programmer describes a protocol once, as a global workflow, and projection compiles it to one local program per agent lifeline. The semantics is based on message sequence charts (MSCs) [15], where events are linearly ordered on each lifeline, and messages connect send events to their matching receive events. This gives a formal basis for visualization, model checking, and runtime verification.

The core language of ZIPPERGEN contains messages, local actions such as tool or function calls, conditionals, loops, and sequential composition. This is enough for many protocols, but it cannot directly express two subprograms that should be unordered while sharing lifelines. Encoding them by sequential composition imposes an order on the shared lifelines that is not part of the protocol.

This paper adds a structured parallel-region operator to ZIPPERGEN. A parallel region consists of several full workflow branches. Branches may contain messages, local actions, conditionals, loops, and nested parallel regions. They may also share lifelines. Projection assigns separate branch-local channel names to messages in different branches, although no channel annotations are written in the source program. It also maps a lifeline that occurs in several branches to one local program with a local interleaving operator over the corresponding branch projections.

The point of the construct is not only to run disjoint workflows side by side, but workflows where branches overlap on some lifelines. Such a lifeline must preserve its own local order, but it should not

Submitted to:
EXPRESS/SOS 2026

force a global order between the branches. Branch-local channel names separate the FIFO queues used by different branches, while the local interleaving operator records that the shared lifeline may execute branch code in an interleaved order.

The main technical difficulty is to make this form of parallelism compatible with endpoint projection and deadlock-freedom. If two branches share lifelines without restriction, some shuffles may create a causal cycle, and such a cycle can become a circular wait after projection. The semantics therefore admits only those parallel shuffles in which the induced causal order is acyclic. It is still nonempty whenever the branch execution sets are nonempty, because executing the branches one after another gives a globally consistent shuffle.

In summary, our contribution is a conservative extension of ZIPPERGEN with source-level parallel regions and shared lifelines. We define the extended projection, the corresponding local interleaving semantics, and the filtered MSC semantics of parallel regions. We prove that projection preserves the workflow semantics and yields deadlock-free distributed programs. The parallel operator is implemented in the open-source ZIPPERGEN prototype, available at `https://zippergen.io`.

Related work. This work builds on MSCs and High-level MSCs (HMSCs) [1, 2, 10, 15]. In the MSC standard, co-regions already relax the order of events on a single lifeline. The parallel regions studied here extend this idea to full workflow branches that may overlap on several lifelines. Parallel and composition operators for MSC-like objects have been studied in several forms. The paper [18] gives a pomset semantics for the MSC'96 parallel operator. Products of MSC languages have been studied in [7], where events are shuffled processwise and shared local events serve as synchronization points. Scenario merging combines partial scenarios by identifying common events [12, 19]. Moreover, causal MSCs relax lifeline order by using causal independence [9]. In contrast, our parallel operator is part of the source language of a projected workflow: shared lifelines are projected to local interleavings, and branch-local channel names keep messages from different branches separate.

Multiparty session types also start from global descriptions and derive local endpoint behaviours [8, 14]. More general projection mechanisms and compositional protocol specifications have been studied in [11, 21]. The paper [5] defines global types as trace languages with sequential, alternative, and unconstrained composition. The latter is interpreted by shuffle, as in our parallel regions. Moreover, controlling dependency cycles is a standard way to deadlock-freedom in message-passing and session-type systems [6, 16, 17, 20].

Choreographic programming treats global descriptions as executable programs rather than only as protocol specifications. Endpoint projection is then used to derive the local code [4, 13, 22, 24]. Asynchronous choreographic models already allow independent communications to be permuted by the semantics. Compositional choreographies add an explicit parallel operator for composing independently developed choreographies [23]. In our setting, the branches are parts of one structured workflow, shared lifelines are handled by local interleaving, and the source language has no partial actions.

Outline. In Section 2, we recall the sequential ZIPPERGEN language. Section 3 introduces the parallel construct with its syntax and semantics. Section 4 gives the extended projection and the local semantics of the projected interleaving operator at shared lifelines. Section 5 proves deadlock-freedom and semantic correctness. We conclude in Section 6.

Acknowledgments. This work was partly supported by the "France 2030" government investment plan managed by ANR, under the reference ANR-23-PEIA-0006.

2 Preliminaries: ZIPPERGEN without Parallel

We recall the ZIPPERGEN syntax and semantics from [3]. A global program describes the coordination of a fixed set of lifelines via messages, local actions, conditionals, and loops. Projection derives a local program for each lifeline, and the Zipper lemma establishes that the distributed execution of these local programs is deadlock-free and matches the global semantics. Section 3 extends this framework with the parallel construct.

2.1 Global programs

Let $\mathbb{A}$ be a finite set of participant *lifelines*, $\mathbb{X}$ a set of typed variables, $\mathbb{F}$ a set of function names, and $\mathbb{C}$ a set of Boolean conditions over $\mathbb{X}$.

Definition 2.1 (Global programs)**.** The set of *global programs* is given by:

$$\begin{aligned} P ::= \varepsilon \mid \texttt{msg}\, A(\vec{x}) \to B(\vec{y}) \mid \texttt{act}\, A : \vec{y} = f(\vec{x}) \mid P_1 \,;\, P_2 \\ \mid \texttt{if}\ \varphi@A\ \texttt{then}\ P_{\mathsf{then}}\ \texttt{else}\ P_{\mathsf{else}} \\ \mid \texttt{while}\ \varphi@A\ \texttt{do}\ P_{\mathsf{body}}\ \texttt{exit}\ P_{\mathsf{exit}} \end{aligned}$$

where $A, B \in \mathbb{A}$ are distinct, $f \in \mathbb{F}$, and $\varphi \in \mathbb{C}$.

In message statements (`msg`), $\vec{x}$ and $\vec{y}$ range over tuples of constants and variables. In action statements (`act`), the input tuple $\vec{x}$ may contain constants and variables, while $\vec{y}$ is a tuple of variables used as assignment targets. The statement $\texttt{msg}\, A(\vec{x}) \to B(\vec{y})$ sends the values of $\vec{x}$ from A to B, where they are received into $\vec{y}$. The statement $\texttt{act}\, A : \vec{y} = f(\vec{x})$ performs the local operation f at lifeline A and stores its results in $\vec{y}$. Lifeline A in $\texttt{if}\ \varphi@A$ and $\texttt{while}\ \varphi@A$ is the *owner* (decider) of the construct.

Definition 2.2 (Lifelines of a program)**.** The set $\mathscr{L}(P)$ of lifelines of P is defined inductively as follows: $\mathscr{L}(\varepsilon) = \emptyset$, $\mathscr{L}(\texttt{msg}\ A(\vec{x}) \to B(\vec{y})) = \{A, B\}$, $\mathscr{L}(\texttt{act}\ A : \vec{y} = f(\vec{x})) = \{A\}$, and $\mathscr{L}(P_1;P_2) = \mathscr{L}(P_1) \cup \mathscr{L}(P_2)$. Similarly, for `if` and `while` the lifelines are the union of owner and branch lifelines.

2.2 MSC semantics

Let CName be a set of *channel names* with a distinguished element main (the top-level name; concrete channel names for parallel regions are defined in Section 3.3). For lifeline $A \in \mathbb{A}$, the (infinite) *local alphabet* Σ_A contains:

- send letters $\texttt{send}\, A(\vec{x}) \to B[c]$ and receive letters $\texttt{recv}\, A(\vec{y}) \leftarrow B[c]$, for $B \in \mathbb{A} \setminus \{A\}$, $c \in \mathsf{CName}$, and payload tuples $\vec{x}, \vec{y}$ of constants and variables;
- local action letters $\texttt{act}\, A : \vec{y} = f(\vec{x})$, where $f \in \mathbb{F}$, $\vec{x}$ ranges over tuples of constants and variables, and $\vec{y}$ over tuples of variables;
- choice letters $\mathsf{if}_\top(\varphi@A), \mathsf{if}_\bot(\varphi@A), \mathsf{while}_\top(\varphi@A), \mathsf{while}_\bot(\varphi@A)$, for each condition $\varphi \in \mathbb{C}$.

Projected programs use control broadcasts to communicate branch and loop decisions to the other lifelines. These broadcasts are modeled as ordinary send/receive letters in the projected MSC semantics. The owner of a construct P of the form $\texttt{if}\ \varphi@A$ or $\texttt{while}\ \varphi@A$ emits $\texttt{send}\, A(b, \mathsf{tag}^P_{\mathsf{ctrl}}) \to C[c]$ to each recipient C, where $b \in \{\top, \bot\}$ is the truth value of the guard and $\mathsf{tag}^P_{\mathsf{ctrl}}$ is a reserved tag unique to P. On the other hand, C contributes a matching $\texttt{recv}\, C(b, \mathsf{tag}^P_{\mathsf{ctrl}}) \leftarrow A[c]$ and branches on b. These messages obey the same FIFO rules as user messages. They are not source syntax. Following [3], we use enriched MSCs that contain these broadcasts and then erase them for the control-erased semantic observation.

Definition 2.3 (Payload matching)**.** For payload tuples $\vec{x}$ (for the sender) and $\vec{y}$ (for the receiver) whose components may be constants or variables, we write $\mathsf{match}(\vec{x},\vec{y})$ if they have the same length and, componentwise, every receiver-side constant is matched by the same sender-side constant, while receiver variables accept values of the appropriate type. This is the same compatibility condition as in [3].

As in [3], all results below are stated for *well-typed* programs. In particular, every message subprogram $\mathtt{msg}\ A(\vec{x}) \to B(\vec{y})$ satisfies $\mathsf{match}(\vec{x},\vec{y})$, every action invocation $f(\vec{x})$ is locally type-correct for its output tuple $\vec{y}$, and every guard $\varphi@A$ is a Boolean expression over values available at lifeline A.

Next, we define message sequence charts (MSCs), as tuples of words that induce a FIFO relation on the induced event set.

Definition 2.4 (FIFO relation)**.** For a tuple $M = (w_A)_{A\in\mathbb{A}}$ with $w_A \in \Sigma_A^*$, let $|w_A|$ be the length of w_A and $w_A[j]$ its jth symbol, using indices starting at 1. The event set of M is

$$E_M := \{(A,j) \mid A \in \mathbb{A},\ 1 \le j \le |w_A|\}.$$

For $A \neq B$ and $c \in \mathsf{CName}$, let $\mathsf{snd}_{A\to B[c]}(M) = (s_1,\dots,s_m)$ be the send events (A,j) whose letter is of the form $\mathtt{send}\ A(\cdot) \to B[c]$, listed in local order. Let $\mathsf{rcv}_{B\leftarrow A[c]}(M) = (r_1,\dots,r_n)$ be the matching sequence on the receiver side, namely the events (B,j) whose letter is of the form $\mathtt{recv}\ B(\cdot) \leftarrow A[c]$. The *FIFO relation* $\lhd_M \subseteq E_M \times E_M$ contains the pairs (s_k,r_k) for all $k \le \min(m,n)$, for all $A \neq B$ and $c \in \mathsf{CName}$.

Definition 2.5 (MSC)**.** A tuple $M = (w_A)_{A\in\mathbb{A}}$ is an *MSC* if:

(1) every receive event is matched in $\lhd_M$;

(2) whenever a send event with payload $\vec{x}$ is matched to a receive event with payload $\vec{y}$, we have $\mathsf{match}(\vec{x},\vec{y})$;

(3) the transitive closure of $\lhd_M \cup \{((A,j),(A,j+1)) \mid A \in \mathbb{A},\ 1 \le j < |w_A|\}$ is a strict partial order, that is, acyclic.

It is a *complete MSC* if additionally every send event is matched in $\lhd_M$.

Definition 2.6 (Control erasure)**.** For a tuple of local words $M = (w_A)_A$, the tuple $\mathsf{erase}_{\mathsf{ctrl}}(M)$ is obtained by removing every control-broadcast send or receive symbol, that is, every send/receive letter carrying some reserved tag $\mathsf{tag}^{Q}_{\mathsf{ctrl}}$, from each w_A.

Control erasure keeps choice letters. In the enriched MSCs generated by projected programs (the only ones to which we apply erasure) each control send is matched with its corresponding control receive, since both carry the reserved tag $\mathsf{tag}^{Q}_{\mathsf{ctrl}}$ and are routed on the same subchannel. For such an M, if M is a complete MSC then $\mathsf{erase}_{\mathsf{ctrl}}(M)$ is again a complete MSC: matched control sends and receives are deleted in pairs; after re-indexing the remaining sends and receives on each directed channel, payload matching is unchanged, and deleting events cannot create a causal cycle. We extend control erasure pointwise to sets of MSCs.

Thus MSCs in this paper may be incomplete: unmatched sends represent messages in transit. Programs without the parallel operator use only channel name main, reducing to one queue per directed pair as in [3]. We next set up the channel-parametric semantics needed by parallel branches.

We use componentwise concatenation of MSC tuples: $M_1 \circ M_2 := (u_A v_A)_{A\in\mathbb{A}}$ for $M_1 = (u_A)_A$ and $M_2 = (v_A)_A$. As in [3], concatenation is stable when the left factor is complete: if M_1 is a complete MSC and M_2 is an MSC, then $M_1 \circ M_2$ is an MSC; if M_2 is complete as well, then $M_1 \circ M_2$ is complete. Conversely, if M is a complete MSC and $M \circ N$ is an MSC, then N is an MSC (and complete when $M \circ N$ is complete).

Let $\mathbf{0} := (\varepsilon)_{A\in\mathbb{A}}$ denote the empty MSC tuple. For a letter $\alpha \in \Sigma_A$, write $\mathsf{one}_A(\alpha)$ for the tuple with word α at A and empty words elsewhere. For a source message $\mathtt{msg}\ A(\vec{x}) \to B(\vec{y})$ and channel name c, write $\mathsf{msg}^c_{A,B}(\vec{x},\vec{y})$ for the complete MSC with local words $\mathtt{send}\ A(\vec{x}) \to B[c]$ at A and $\mathtt{recv}\ B(\vec{y}) \leftarrow A[c]$ at B, and ε elsewhere.

For a control construct P with owner A, the *recipient set* $\mathscr{R}(P)$ is given by:

$$\mathscr{R}(\mathtt{if}\ \varphi@A\ \mathtt{then}\ P_{\mathsf{then}}\ \mathtt{else}\ P_{\mathsf{else}}) = (\mathscr{L}(P_{\mathsf{then}}) \cup \mathscr{L}(P_{\mathsf{else}})) \setminus \{A\},$$
$$\mathscr{R}(\mathtt{while}\ \varphi@A\ \mathtt{do}\ P_{\mathsf{body}}\ \mathtt{exit}\ P_{\mathsf{exit}}) = (\mathscr{L}(P_{\mathsf{body}}) \cup \mathscr{L}(P_{\mathsf{exit}})) \setminus \{A\}.$$

Fix once and for all a global order on the lifelines. Each finite recipient set is enumerated by restricting this order. Let Q be a control construct with owner A, and let $b \in \{\top,\bot\}$. Recall that, for an if-construct the owner choice letter is $\mathsf{if}_\top(\varphi@A)$ or $\mathsf{if}_\bot(\varphi@A)$, and for a while-construct it is $\mathsf{while}_\top(\varphi@A)$ or $\mathsf{while}_\bot(\varphi@A)$. The *decision block* $\mathsf{Dec}^c_Q(b)$ is the tuple of local words that contains, on A, the owner choice letter for truth value b followed by the control sends $\mathtt{send}\ A(b,\mathsf{tag}^Q_{\mathsf{ctrl}}) \to C[c]$ for all $C \in \mathscr{R}(Q)$ in the fixed order. On each recipient C, it contains the matching receive $\mathtt{recv}\ C(b,\mathsf{tag}^Q_{\mathsf{ctrl}}) \leftarrow A[c]$; all other local words are empty. Each decision block is a complete MSC, and erasing controls leaves only the owner choice letter.

The internal enriched semantics retains the ordinary source steps together with the control broadcasts inserted by projection.

Definition 2.7 (Internal enriched semantics)**.** For a global program P and channel name $c \in \mathsf{CName}$, the *internal enriched semantics* $\mathsf{Enr}^c(P)$ is defined as follows. For

$$I = \mathtt{if}\ \varphi@A\ \mathtt{then}\ P_{\mathsf{then}}\ \mathtt{else}\ P_{\mathsf{else}} \quad \text{and} \quad W = \mathtt{while}\ \varphi@A\ \mathtt{do}\ P_{\mathsf{body}}\ \mathtt{exit}\ P_{\mathsf{exit}},$$

we let

$$\begin{aligned}
\mathsf{Enr}^c(\varepsilon) &= \{\mathbf{0}\},\\
\mathsf{Enr}^c(\mathtt{act}\ A : \vec{y} = f(\vec{x})) &= \{\mathsf{one}_A(\mathtt{act}\ A : \vec{y} = f(\vec{x}))\},\\
\mathsf{Enr}^c(\mathtt{msg}\ A(\vec{x}) \to B(\vec{y})) &= \{\mathsf{msg}^c_{A,B}(\vec{x},\vec{y})\},\\
\mathsf{Enr}^c(P_1;P_2) &= \{M_1 \circ M_2 \mid M_1 \in \mathsf{Enr}^c(P_1),\ M_2 \in \mathsf{Enr}^c(P_2)\},\\
\mathsf{Enr}^c(I) &= \{\mathsf{Dec}^c_I(\top) \circ M \mid M \in \mathsf{Enr}^c(P_{\mathsf{then}})\}\\
&\quad \cup \{\mathsf{Dec}^c_I(\bot) \circ M \mid M \in \mathsf{Enr}^c(P_{\mathsf{else}})\}.
\end{aligned}$$

For the while program W, let E^{while}_0 contain the executions

$$\mathsf{Dec}^c_W(\bot) \circ M_{\mathsf{exit}}$$

with $M_{\mathsf{exit}} \in \mathsf{Enr}^c(P_{\mathsf{exit}})$, and let E^{while}_{k+1} contain the executions

$$\mathsf{Dec}^c_W(\top) \circ M_{\mathsf{body}} \circ M$$

with $M_{\mathsf{body}} \in \mathsf{Enr}^c(P_{\mathsf{body}})$ and $M \in E^{\mathsf{while}}_k$. Then $\mathsf{Enr}^c(W) = \bigcup_{k\geq 0} E^{\mathsf{while}}_k$. The control-erased semantic observation is $[\![P]\!]^c := \mathsf{erase}_{\mathsf{ctrl}}(\mathsf{Enr}^c(P))$.

The rule for the parallel operator is added in Definition 3.4. We write $[\![P]\!]_{\mathsf{src}}$ for $[\![P]\!]^{\mathsf{main}}$, the source-level observation used in the correctness theorem. Despite the subscript, this observation still carries the compiler-assigned branch channel names, which distinguish FIFO queues in the MSC representation.

Unlike the sequential core of ZIPPERGEN, the parallel extension uses this projection-aligned internal enriched semantics before control erasure. Section 3.4 explains why parallel shuffles must be filtered before projected control broadcasts are erased.

2.3 Projection and local programs

For a fixed lifeline A, the projection $\pi_A(P)$ derives a *local program* for A. The grammar of local programs for A is given as follows:

$$\begin{aligned} S ::= {} & \varepsilon \mid \texttt{send}\, A(\vec{x}) \to B[c] \mid \texttt{recv}\, A(\vec{y}) \leftarrow B[c] \mid \texttt{act}\, A : \vec{y} = f(\vec{x}) \mid S_1 \,;\, S_2 \\ & \mid \texttt{if}\ \varphi@A\ \texttt{then}\ S_{\mathsf{then}}\ \texttt{else}\ S_{\mathsf{else}} \mid \texttt{if}\ A(\vec{y}) \leftarrow B[c]\ \texttt{then}\ S_{\mathsf{then}}\ \texttt{else}\ S_{\mathsf{else}} \\ & \mid \texttt{while}\ \varphi@A\ \texttt{do}\ S_{\mathsf{body}}\ \texttt{exit}\ S_{\mathsf{exit}} \mid \texttt{while}\ A(\vec{y}) \leftarrow B[c]\ \texttt{do}\ S_{\mathsf{body}}\ \texttt{exit}\ S_{\mathsf{exit}} \end{aligned}$$

The local constructs $\texttt{if}\ \varphi@A$ and $\texttt{while}\ \varphi@A$ are used at the lifeline that evaluates the guard and records the corresponding choice letter; this is the meaning of the subscript own below. Recipient-side constructs $\texttt{if}\ A(\vec{y}) \leftarrow B[c]$ and $\texttt{while}\ A(\vec{y}) \leftarrow B[c]$ use a non-empty tuple $\vec{y}$ of constants and variables. Writing $\vec{y} = (y_1, y_2, \ldots)$, the first component y_1 is a Boolean variable for the truth value (later, projected control receives use the second component to carry the construct tag). Moreover, $c \in \mathsf{CName}$: the projection rules below determine the channel name used at each send and receive.

Definition 2.8 (Local trace semantics)**.** For a local program S at lifeline A, the complete local trace language $\langle\!\langle S \rangle\!\rangle \subseteq \Sigma_A^*$ is defined inductively. For

$$\begin{aligned} I_{\mathsf{own}} &= \texttt{if}\ \varphi@A\ \texttt{then}\ S_{\mathsf{then}}\ \texttt{else}\ S_{\mathsf{else}}, \\ I_{\mathsf{recv}} &= \texttt{if}\ A(\vec{y}) \leftarrow B[c]\ \texttt{then}\ S_{\mathsf{then}}\ \texttt{else}\ S_{\mathsf{else}}, \\ W_{\mathsf{own}} &= \texttt{while}\ \varphi@A\ \texttt{do}\ S_{\mathsf{body}}\ \texttt{exit}\ S_{\mathsf{exit}}, \\ W_{\mathsf{recv}} &= \texttt{while}\ A(\vec{y}) \leftarrow B[c]\ \texttt{do}\ S_{\mathsf{body}}\ \texttt{exit}\ S_{\mathsf{exit}}, \end{aligned}$$

the semantics is given as follows:

$$\begin{aligned} \langle\!\langle \varepsilon \rangle\!\rangle &= \{\varepsilon\}, \\ \langle\!\langle \texttt{send}\, A(\vec{x}) \to B[c] \rangle\!\rangle &= \{\texttt{send}\, A(\vec{x}) \to B[c]\}, \\ \langle\!\langle \texttt{recv}\, A(\vec{y}) \leftarrow B[c] \rangle\!\rangle &= \{\texttt{recv}\, A(\vec{y}) \leftarrow B[c]\}, \\ \langle\!\langle \texttt{act}\, A : \vec{y} = f(\vec{x}) \rangle\!\rangle &= \{\texttt{act}\, A : \vec{y} = f(\vec{x})\}, \\ \langle\!\langle S_1 ; S_2 \rangle\!\rangle &= \{uv \mid u \in \langle\!\langle S_1 \rangle\!\rangle,\ v \in \langle\!\langle S_2 \rangle\!\rangle\}, \\ \langle\!\langle I_{\mathsf{own}} \rangle\!\rangle &= \{\mathsf{if}_\top(\varphi@A)u \mid u \in \langle\!\langle S_{\mathsf{then}} \rangle\!\rangle\} \cup \{\mathsf{if}_\bot(\varphi@A)u \mid u \in \langle\!\langle S_{\mathsf{else}} \rangle\!\rangle\}. \end{aligned}$$

For a receive-guarded conditional with owner B, we write $\vec{y}_\top$ and $\vec{y}_\bot$ for the received payload tuples with first component $\top$ or $\bot$ and all other components unchanged, respectively. With this, we define

$$\begin{aligned} \langle\!\langle I_{\mathsf{recv}} \rangle\!\rangle = {} & \{\texttt{recv}\, A(\vec{y}_\top) \leftarrow B[c]\, u \mid u \in \langle\!\langle S_{\mathsf{then}} \rangle\!\rangle\} \\ & \cup \{\texttt{recv}\, A(\vec{y}_\bot) \leftarrow B[c]\, u \mid u \in \langle\!\langle S_{\mathsf{else}} \rangle\!\rangle\}. \end{aligned}$$

Local while loops at the guard lifeline are finite iterations. Let T_0 contain the words $\mathsf{while}_\bot(\varphi@A)v$ with $v \in \langle\!\langle S_{\mathsf{exit}} \rangle\!\rangle$, and let T_{k+1} contain the words $\mathsf{while}_\top(\varphi@A)uw$ with $u \in \langle\!\langle S_{\mathsf{body}} \rangle\!\rangle$ and $w \in T_k$. Then $\langle\!\langle W_{\mathsf{own}} \rangle\!\rangle = \bigcup_{k \geq 0} T_k$. For the receive-guarded loop, let $r_b = \texttt{recv}\, A(\vec{y}_b) \leftarrow B[c]$ for $b \in \{\top, \bot\}$, define $R_0 = \{r_\bot v \mid v \in \langle\!\langle S_{\mathsf{exit}} \rangle\!\rangle\}$ and $R_{k+1} = \{r_\top uw \mid u \in \langle\!\langle S_{\mathsf{body}} \rangle\!\rangle,\ w \in R_k\}$, and set $\langle\!\langle W_{\mathsf{recv}} \rangle\!\rangle = \bigcup_{k \geq 0} R_k$. We write $\langle\!\langle S \rangle\!\rangle_{\mathsf{pref}} := \{u \mid \exists v.\ uv \in \langle\!\langle S \rangle\!\rangle\}$ for the prefix closure of $\langle\!\langle S \rangle\!\rangle$. Hence $\langle\!\langle S \rangle\!\rangle \subseteq \langle\!\langle S \rangle\!\rangle_{\mathsf{pref}}$, and $\varepsilon \in \langle\!\langle S \rangle\!\rangle_{\mathsf{pref}}$ whenever $\langle\!\langle S \rangle\!\rangle$ is nonempty.

The local trace language generates candidate words for one lifeline. It does not by itself impose communication well-formedness: a send letter in one local word need not have a matching receive until the tuple of local words is checked by the MSC conditions in the distributed semantics below.

3 Parallel Regions

The parallel construct adds a parallel operator to the global program language. Branches are full programs that may share lifelines. Channel names assigned by the compiler separate their messages without exposing new source-level syntax. This section defines the syntax of the new construct (Section 3.1), a motivating example (Section 3.2), the channel-name assignment (Section 3.3), and the internal enriched semantics used to interpret parallel shuffles (Section 3.4).

3.1 Syntax

A parallel region has finitely many branches, each of which is an ordinary global program. It adds no new message, action, or control statements, and branches may share lifelines.

Definition 3.1 (Extended global programs)**.** The grammar of Definition 2.1 is extended with:

$$P \ ::= \ \ldots \ \mid \ \mathsf{parallel}(P_1, \ldots, P_n) \qquad (n \geq 2)$$

Each P_i is a *branch* and is itself a full global program (in particular, branches may contain `if`, `while`, and nested parallel). The lifelines of a parallel region are $\mathscr{L}(\mathsf{parallel}(P_1, \ldots, P_n)) = \mathscr{L}(P_1) \cup \ldots \cup \mathscr{L}(P_n)$.

If a lifeline occurs in several branches, projection will combine the corresponding local branch programs by the local interleaving operator introduced in Section 4.

3.2 Example: command center

Program 1 sketches a personal command center. One stream classifies email, another stream handles chat commands from the owner. The streams are independent, but they share lifelines: the Dispatcher classifies requests, Calendar performs calendar actions, and User resolves human-approval actions. The email stream also uses Writer and Mailbox for drafting and mailbox operations. The outer block is the parallel-region construct, and `skip` denotes the empty program ε of Definition 2.1. Here, "..." is a placeholder for omitted route cases.

The service that motivates the example is meant to keep running. The semantics in this paper, however, is a semantics of finite MSCs. Program 1 should therefore be read as one finite service window: each loop contributes some finite number of iterations and then takes its exit. An always-on deployment is modeled by repeated finite windows, or by reasoning about arbitrary finite prefixes of the service.

The parallel structure matters even in one finite window. A sequential encoding would have to decide, for example, whether all pending email is handled before any chat command, or conversely. That order is not part of the intended coordination. In the parallel program, Mailbox and Chat can make progress independently, while shared lifelines such as Dispatcher, Calendar, and User locally interleave the work addressed to them.

At the same time, not every syntactic interleaving of the two branches is globally meaningful. The chat branch may send a request from Calendar to User and then a confirmation from User back to Calendar; other email routes may pass through Calendar, User, Writer, and Mailbox. When these branch-local orders are interleaved on shared lifelines, some tuples can create circular causal dependencies. Such tuples are syntactic shuffles, but they are not MSCs, so they are not internal enriched executions. The semantics below accepts the globally consistent command-center schedules and filters out only the inconsistent ones.

The parallel construct treats writes at a shared lifeline as ordinary interleaved local actions. If the final value of a local variable matters, the programmer must ensure that the updates are independent, commutative, or intentionally ordered by messages.

Program 1: One finite command-center window.

```
parallel(
    while email_window@Mailbox do
        msg Mailbox(email) -> Dispatcher(email) ;
        act Dispatcher : route = classify_email(email) ;
        if route = "reply"@Dispatcher then
            msg Dispatcher(email) -> Writer(email) ;
            act Writer : reply = draft_reply(email) ;
            msg Writer(reply) -> User(reply) ;
            act User : edit = approve_or_edit(reply) ;
            if edit@User then
                msg User(edit) -> Mailbox(edit) ;
                act Mailbox : status = save_draft(edit)
            else
                skip
        else
            ...
    exit
        skip,

    while chat_window@Chat do
        msg Chat(cmd) -> Dispatcher(cmd) ;
        act Dispatcher : chat_route = classify_chat(cmd) ;
        if chat_route = "schedule"@Dispatcher then
            msg Dispatcher(cmd) -> Calendar(cmd) ;
            act Calendar : event = extract_event(cmd) ;
            msg Calendar(event) -> User(event) ;
            act User : ok = confirm_event(event) ;
            if ok@User then
                msg User(event) -> Calendar(event) ;
                act Calendar : status = create_event(event) ;
                msg Calendar(status) -> Chat(status)
            else
                msg User(ok) -> Chat(ok)
        else
            ...
    exit
        skip
)
```

3.3 Channel Names

Send/receive actions in different branches of the same parallel region may use the same source-level channel $A \to B$. The compiler therefore refines each directed FIFO channel $A \to B$ by an internal channel name $c \in \mathsf{CName}$, yielding subchannels (A, B, c). FIFO order is maintained per subchannel.

Definition 3.2 (Channel names)**.** Each parallel occurrence in a program carries a distinct *occurrence label* o (concretely, its position in the program syntax tree; the prototype uses the node identity). CName is the set of finite paths whose steps are pairs $o.i$ of an occurrence label o and a positive branch index i. The root path is main, and if $c \in \mathsf{CName}$ then $c.o.i$ appends the step for branch i of the occurrence

o. Top-level sends and receives in branch i of a parallel occurrence o reached under current channel c carry $c.o.i$. A nested parallel extends the path further. Because distinct occurrences carry distinct labels, branches of different parallel occurrences never share a channel.

Channel names are not part of the source language. The set CName contains all possible names. Each compiled program uses a finite subset. Programs without parallel use only main.

Control tags serve a different purpose. As in the base ZIPPERGEN proof, each syntactic occurrence Q of an `if`- or `while`-construct induces a reserved tag $\mathsf{tag}^{Q}_{\mathsf{ctrl}}$, and distinct occurrences use distinct reserved tags. Tags distinguish control constructs within a channel, whereas channel names separate concurrent branch queues.

For a program P reached under current channel c, let $\mathsf{Ch}^c(P)$ be the finite set of branch channel names generated by parallel constructs in P, including P itself if it is parallel:

$$\begin{aligned}
\mathsf{Ch}^c(\varepsilon) = \mathsf{Ch}^c(\texttt{msg}\,A(\vec{x}) \to B(\vec{y})) = \mathsf{Ch}^c(\texttt{act}\,A : \vec{y} = f(\vec{x})) &= \emptyset,\\
\mathsf{Ch}^c(P_1;P_2) &= \mathsf{Ch}^c(P_1) \cup \mathsf{Ch}^c(P_2),\\
\mathsf{Ch}^c(\texttt{if}\ \varphi@A\ \texttt{then}\ P_{\mathsf{then}}\ \texttt{else}\ P_{\mathsf{else}}) &= \mathsf{Ch}^c(P_{\mathsf{then}}) \cup \mathsf{Ch}^c(P_{\mathsf{else}}),\\
\mathsf{Ch}^c(\texttt{while}\ \varphi@A\ \texttt{do}\ P_{\mathsf{body}}\ \texttt{exit}\ P_{\mathsf{exit}}) &= \mathsf{Ch}^c(P_{\mathsf{body}}) \cup \mathsf{Ch}^c(P_{\mathsf{exit}}),\\
\mathsf{Ch}^c(\mathsf{parallel}^o(P_1,\ldots,P_n)) &= \bigcup_{i=1}^{n} \big(\{c.o.i\} \cup \mathsf{Ch}^{c.o.i}(P_i)\big).
\end{aligned}$$

Every element of $\mathsf{Ch}^c(P)$ strictly extends c, so $c \notin \mathsf{Ch}^c(P)$; in particular $c.o.i \notin \mathsf{Ch}^{c.o.i}(P_i)$. Because distinct parallel occurrences carry distinct labels, two separation properties hold. *Sibling disjointness:* for a fixed occurrence o, the sets $C_{o,i} = \{c.o.i\} \cup \mathsf{Ch}^{c.o.i}(P_i)$ for its branches $i = 1,\ldots,n$ are pairwise disjoint, since they extend the distinct prefixes $c.o.i$. *Per-name origin:* every channel name other than main is generated by a unique parallel-branch occurrence—the one named by its final step $o.i$—and, as FIFO matches only equal names, the matching event of a send or receive on a name stays within that branch occurrence. We call these properties *channel-origin uniqueness*. For *different* occurrences the sets $C_{o,i}$ need not be disjoint: a name from a nested branch lies in the enclosing branch's set as well. The proofs use only sibling disjointness and per-name origin, both of which hold globally.

Control broadcasts are treated like ordinary messages with respect to channel names: they inherit the current channel name. Thus, if a construct Q of the form `if` $\varphi@A$ or `while` $\varphi@A$ is projected under channel c, every control send $\texttt{send}\,A(b, \mathsf{tag}^{Q}_{\mathsf{ctrl}}) \to C[c]$ uses subchannel c. Directly inside branch i of a parallel occurrence o reached under current channel c, the current channel is $c.o.i$, and inside a nested parallel branch it is extended again. The reserved tag $\mathsf{tag}^{Q}_{\mathsf{ctrl}}$ identifies the particular `if`/`while` construct Q within that channel. The two mechanisms therefore serve distinct roles: the occurrence/path-based channel name separates parallel branches, while the construct-based control tag distinguishes individual control constructs within a branch.

3.4 Semantics

The internal enriched semantics of a parallel region consists of the complete MSCs that arise as shuffles of one enriched execution from each branch, evaluated under the branch-local channel names assigned in Section 3.3. The source observation is obtained by applying control erasure to this enriched semantics, as in Definition 2.7.

Definition 3.3 (Shuffle)**.** Let $M = (w_A)_{A\in\mathbb{A}}$ and $M_i = (u^i_A)_{A\in\mathbb{A}}$ for $1 \le i \le n$ be tuples of local words. We say that M is a *shuffle* of $M_1,\ldots,M_n$ if, for each $A \in \mathbb{A}$, the word w_A is an interleaving of $u^1_A,\ldots,u^n_A$, that

is, there exist order-preserving injections $\phi_i : \{1,\dots,|u_A^i|\} \to \{1,\dots,|w_A|\}$ with pairwise disjoint ranges that together cover $\{1,\dots,|w_A|\}$, such that $w_A[\phi_i(k)] = u_A^i[k]$ for all i,k.

Definition 3.4 (Internal semantics of parallel)**.** Let $P = \mathsf{parallel}^o(P_1,\dots,P_n)$ with occurrence label o. Its internal enriched semantics under current channel c is

$$\mathsf{Enr}^c(P) = \Big\{ M \;\Big|\; M \text{ is a shuffle of some } M_1 \in \mathsf{Enr}^{c.o.1}(P_1),\dots,M_n \in \mathsf{Enr}^{c.o.n}(P_n) \text{ and } M \text{ is a complete MSC} \Big\}.$$

Note that, by Definition 3.2, $c \notin \mathsf{Ch}^c(P)$ for every program P. Together with Definition 2.7, we now have the complete internal enriched semantics of extended programs. The control-erased observation remains $[\![P]\!]^c = \mathsf{erase}_{\mathsf{ctrl}}(\mathsf{Enr}^c(P))$.

This is a filtered shuffle semantics. The syntactic shuffle relation describes all branch-local interleavings, while the semantic rule keeps exactly those enriched interleavings whose induced tuple is a complete MSC. Hence a cyclic local ordering such as "both sides receive before sending" is not an execution. For instance, with branches $\mathtt{msg}\ A \to B$ and $\mathtt{msg}\ B \to A$, each branch is acyclic on its own. But a local shuffle may put, on A, the receive from B before the send to B and, on B, the receive from A before the send to A. The two local orders and the two message edges form a causal cycle, so this syntactic shuffle is rejected by the MSC filter.

There is a second, more specifically projected reason to apply the MSC filter before control erasure. Validity of a parallel shuffle may depend on causal edges introduced by projected control broadcasts. Erasing these broadcasts before applying the MSC filter would accept tuples that cannot arise as projected executions. Consider

$$P = \mathsf{parallel}^o(I, \mathtt{msg}\ B \to A) \qquad \text{where} \qquad I = \mathtt{if}\ \varphi@A\ \mathtt{then}\ \mathtt{act}\ B : y = f()\ \mathtt{else}\ \mathtt{act}\ B : y = g().$$

Projection emits a control message from A to B in either branch. In the true execution considered here, the message carries the value $\top$. A local shuffle may put, on A, the receive of the $B \to A$ message before the owner choice and its control send, and, on B, the control receive and selected action before the $B \to A$ send. After erasing controls, this control-erased tuple has no causal cycle: it contains the $B \to A$ message together with local choice/action order. Before erasure, however, using the branch channels $\mathsf{main}.o.1$ and $\mathsf{main}.o.2$ from Section 3.3, the control edge creates the cycle

$$\begin{aligned} \mathtt{send}\ B \to A[\mathsf{main}.o.2] &\longrightarrow \mathtt{recv}\ A \leftarrow B[\mathsf{main}.o.2] \\ &\longrightarrow \mathtt{send}\ A(\top, \mathsf{tag}^I_{\mathsf{ctrl}}) \to B[\mathsf{main}.o.1] \\ &\longrightarrow \mathtt{recv}\ B(\top, \mathsf{tag}^I_{\mathsf{ctrl}}) \leftarrow A[\mathsf{main}.o.1] \\ &\longrightarrow \mathtt{send}\ B \to A[\mathsf{main}.o.2], \end{aligned}$$

where arrows abbreviate message edges or local-order paths. Thus the tuple is not a distributed execution. This is why parallel composition filters enriched tuples first and erases control broadcasts only afterwards.

The MSC filter does not make a nonempty family of branches vacuous. If each $\mathsf{Enr}^{c.o.i}(P_i)$ is nonempty, choose $M_i \in \mathsf{Enr}^{c.o.i}(P_i)$ for every branch and serialize the branches in the order $1,\dots,n$: on every lifeline A, take the word of branch 1 at A, followed by the word of branch 2 at A, and so on, omitting empty words. Every cross-branch local edge then goes from a smaller branch index to a larger one. Since branch channel sets are disjoint, FIFO matching stays within the corresponding branch block. A causal cycle would therefore either stay inside one branch, contradicting that M_i is an MSC, or strictly increase the branch index and never decrease it. Payload matching and completeness are inherited from the branch MSCs, so the serialized shuffle belongs to $\mathsf{Enr}^c(P)$.

4 Projection and Local Semantics

To project a parallel region, we extend the local program grammar with a local interleaving operator. The projection function carries the current channel name as a superscript.

The local program grammar is extended with:

$$S \;::=\; \ldots \mid \mathsf{parloc}(S_1,\ldots,S_n)$$

where $\mathsf{parloc}(S_1,\ldots,S_n)$ interleaves the branch-local programs while preserving the internal order of each component. Each component S_j carries the channel name of its branch, which the runtime uses to route sends and receives to the correct queue.

The projection therefore carries the current channel c as a superscript, written $\pi_A^c(P)$ (recall that $c \notin \mathsf{Ch}^c(P)$ according to Definition 3.2). The top-level call uses $c = \mathsf{main}$.

Using the global lifeline order fixed above, for a control construct Q with owner B and truth value $b \in \{\top,\bot\}$, let $\mathsf{bcast}^c_{B,Q}(b)$ denote the local program

$$\texttt{send}\; B(b,\mathsf{tag}^Q_{\mathsf{ctrl}}) \to C_1[c]\,;\; \ldots\; ;\; \texttt{send}\; B(b,\mathsf{tag}^Q_{\mathsf{ctrl}}) \to C_m[c],$$

where $\mathscr{R}(Q) = \{C_1,\ldots,C_m\}$ in that order. If $\mathscr{R}(Q) = \emptyset$, this sequence is ε.

Definition 4.1 (Full projection)**.** The projection $\pi_A^c(P)$ is defined by structural induction on P as follows:

$$\begin{aligned}
\pi_A^c(\varepsilon) &= \varepsilon,\\
\pi_A^c(\texttt{act}\; A : \vec{y} = f(\vec{x})) &= \texttt{act}\; A : \vec{y} = f(\vec{x}),\\
\pi_A^c(\texttt{act}\; B : \vec{y} = f(\vec{x})) &= \varepsilon \quad (A \neq B),\\
\pi_A^c(\texttt{msg}\; A(\vec{x}) \to B(\vec{y})) &= \texttt{send}\; A(\vec{x}) \to B[c],\\
\pi_A^c(\texttt{msg}\; B(\vec{x}) \to A(\vec{y})) &= \texttt{recv}\; A(\vec{y}) \leftarrow B[c] \quad (A \neq B),\\
\pi_A^c(\texttt{msg}\; B(\vec{x}) \to C(\vec{y})) &= \varepsilon \quad (A \notin \{B,C\}),\\
\pi_A^c(P_1;P_2) &= \pi_A^c(P_1);\pi_A^c(P_2)
\end{aligned}$$

For

$$I = \texttt{if}\; \varphi@B\; \texttt{then}\; P_{\mathsf{then}}\; \texttt{else}\; P_{\mathsf{else}},$$

control broadcasts use channel name c:

$$\pi_A^c(I) \;=\; \begin{cases}
\texttt{if}\; \varphi@A\; \texttt{then}\; \mathsf{bcast}^c_{A,I}(\top);\, \pi_A^c(P_{\mathsf{then}})\; \texttt{else}\; \mathsf{bcast}^c_{A,I}(\bot);\, \pi_A^c(P_{\mathsf{else}}) & \text{if } A = B,\\
\texttt{if}\; A(z,\mathsf{tag}^I_{\mathsf{ctrl}}) \leftarrow B[c]\; \texttt{then}\; \pi_A^c(P_{\mathsf{then}})\; \texttt{else}\; \pi_A^c(P_{\mathsf{else}}) & \text{if } A \in \mathscr{R}(I),\\
\varepsilon & \text{otherwise.}
\end{cases}$$

For

$$W = \texttt{while}\; \varphi@B\; \texttt{do}\; P_{\mathsf{body}}\; \texttt{exit}\; P_{\mathsf{exit}},$$

we set

$$\pi_A^c(W) \;=\; \begin{cases}
\texttt{while}\; \varphi@A\; \texttt{do}\; \mathsf{bcast}^c_{A,W}(\top);\, \pi_A^c(P_{\mathsf{body}})\; \texttt{exit}\; \mathsf{bcast}^c_{A,W}(\bot);\, \pi_A^c(P_{\mathsf{exit}}) & \text{if } A = B,\\
\texttt{while}\; A(z,\mathsf{tag}^W_{\mathsf{ctrl}}) \leftarrow B[c]\; \texttt{do}\; \pi_A^c(P_{\mathsf{body}})\; \texttt{exit}\; \pi_A^c(P_{\mathsf{exit}}) & \text{if } A \in \mathscr{R}(W),\\
\varepsilon & \text{otherwise.}
\end{cases}$$

In the recipient cases, z is a fresh Boolean variable. For $P = \mathsf{parallel}^o(P_1,\dots,P_n)$ with occurrence label o, set $c_i = c.o.i$ and $S_A^i = \pi_A^{c_i}(P_i)$:

$$\pi_A^c(P) \;=\; \mathsf{parloc}(S_A^1,\dots,S_A^n).$$

The local semantics of parloc is an interleaving of its component traces.

Definition 4.2 (parloc local trace semantics)**.** For the extended local grammar we extend the notation of Definition 2.8, using tagged traces as proof witnesses. The tagged complete local trace language $\langle\!\langle S \rangle\!\rangle^\dagger$ is the ordinary complete local trace language, except that each parloc node records the component from which a letter came. Atomic programs do not add a tag by themselves, but tags are added when their letters are emitted by an enclosing parloc node. The sequential, conditional, and loop cases are the ordinary ones applied recursively. The only new rule is:

$$\begin{aligned}&\langle\!\langle \mathsf{parloc}(S_1,\dots,S_n) \rangle\!\rangle^\dagger \\ &\quad = \left\{ v^\dagger \;\middle|\; \exists w_1^\dagger \in \langle\!\langle S_1 \rangle\!\rangle^\dagger,\dots,w_n^\dagger \in \langle\!\langle S_n \rangle\!\rangle^\dagger \colon v^\dagger \text{ is a tagged interleaving of } w_1^\dagger,\dots,w_n^\dagger \right\}.\end{aligned}$$

Here a tagged interleaving is an ordinary shuffle in which each emitted letter is additionally marked by its component $j \in \{1,\dots,n\}$. The forget map forget removes these marks. They are proof witnesses, not runtime letters. These component tags are distinct from the reserved control tags $\mathsf{tag}^Q_{\mathsf{ctrl}}$ carried by compiler-inserted control messages. For example, if S_1 has local trace $\alpha\beta$ and S_2 has local trace γ, then $\langle 1,\alpha\rangle\langle 2,\gamma\rangle\langle 1,\beta\rangle$ is a tagged trace of $\mathsf{parloc}(S_1,S_2)$, and forget maps it to $\alpha\gamma\beta$. We set

$$\langle\!\langle S \rangle\!\rangle := \mathsf{forget}(\langle\!\langle S \rangle\!\rangle^\dagger) \qquad \text{and} \qquad \langle\!\langle S \rangle\!\rangle_{\mathsf{pref}} := \mathsf{forget}(\langle\!\langle S \rangle\!\rangle^\dagger_{\mathsf{pref}}),$$

where $\langle\!\langle S \rangle\!\rangle^\dagger_{\mathsf{pref}}$ is the prefix closure of $\langle\!\langle S \rangle\!\rangle^\dagger$.

Definition 4.3 (Distributed program and semantics)**.** A *distributed program* $\mathscr{D} = (S_A)_{A\in\mathbb{A}}$ assigns one local program from the extended grammar to each lifeline. Its *complete semantics* and *prefix semantics* are, respectively:

$$\begin{aligned}[\![\mathscr{D}]\!] &:= \left\{ (w_A)_A \;\middle|\; w_A \in \langle\!\langle S_A \rangle\!\rangle \text{ for all } A,\ (w_A)_A \text{ is a complete MSC} \right\},\\ [\![\mathscr{D}]\!]_{\mathsf{pref}} &:= \left\{ (w_A)_A \;\middle|\; w_A \in \langle\!\langle S_A \rangle\!\rangle_{\mathsf{pref}} \text{ for all } A,\ (w_A)_A \text{ is an MSC} \right\}.\end{aligned}$$

Note that we have $[\![\mathscr{D}]\!] \subseteq [\![\mathscr{D}]\!]_{\mathsf{pref}}$.

For a well-typed extended program P and a current channel c, write $D_P^c := \big(\pi_A^c(P)\big)_{A\in\mathbb{A}}$. The top-level projected distributed program is $D_P := D_P^{\mathsf{main}}$.

Scheduler & Implementation. After projection, a parallel region is run by a local scheduler at each lifeline. The parallel operator is implemented this way in a ZIPPERGEN prototype[1], an open-source Python framework for structured multi-agent coordination. There is one sequential executor per lifeline, i.e., parallel branches do not create independent copies. When the executor reaches $\mathsf{parloc}(S_1,\dots,S_n)$, it stores one residual program per component and executes a round-robin policy: starting from the component after the one chosen last, it examines the residuals in order. A send, a local action, and a choice step are always enabled and executed immediately. A receive is enabled only when the corresponding

[1] https://zippergen.io

subchannel already contains a message. If it does not, the scheduler skips to the next component. Thus a blocked receive in one branch does not force the whole shared lifeline to wait. The executor leaves the parloc node once every component has finished its local program. The MSC filter is a denotational characterization of the complete executions admitted by the internal enriched semantics. An implementation does not need to test MSC-validity in advance: receives are enabled only when the corresponding message is available, so cyclic receive-before-send patterns cannot be produced as operational runs. The formal correspondence assumes fairness: an enabled component may be delayed, but not forever.

5 Correctness

We extend the two main results of [3], deadlock-freedom and semantic correctness, to programs with filtered parallel regions. The internal enriched semantics $\mathsf{Enr}^c(P)$ already contains the same decision blocks that projection emits. The control-erased semantic observation is obtained only at the end, by erasing those compiler-inserted control broadcasts. With this convention, the if/while cases of the correctness proof are the cases of [3] with the current channel name carried along: a complete decision block is concatenated before the selected continuation. The parallel case applies the induction hypotheses branchwise and lets the MSC filter check the enriched shuffle.

We write $u \preceq v$ for the prefix order on words over any alphabet, i.e., if there exists a word w such that $uw = v$. In particular, this applies to ordinary local words as well as tagged words. For tuples $U = (u_A)_A$ and $V = (v_A)_A$ of local words, we write $U \preceq V$ if $u_A \preceq v_A$ for every lifeline A. Thus, when M and M' are MSCs represented as tuples of local words, $M \preceq M'$ means componentwise prefix.

Definition 5.1 (Zipper postcondition)**.** Let P be a well-typed extended program, and fix a channel name c. For each lifeline A, write $S_A = \pi_A^c(P)$ for its projection. For tuples of local words $U = (u_A)_A$, $\bar{U} = (\bar{u}_A)_A$, and $V = (v_A)_A$, we write

$$\mathsf{ZipPost}_P^c(U, \bar{U}, V)$$

if the following hold:

(1) for every A: $u_A \preceq \bar{u}_A$, $\quad \bar{u}_A \in \langle\!\langle S_A \rangle\!\rangle$, $\quad$ and $(v_A \neq \varepsilon \implies \bar{u}_A = u_A)$;

(2) $\bar{U}$ is a complete MSC;

(3) $(u_A v_A)_A$ is an MSC;

(4) $(\bar{u}_A v_A)_A$ is an MSC.

This is exactly the Zipper postcondition of [3], except that projections are parameterized by a current channel name c. In particular, condition 2 requires $\bar{U}$ to be a complete MSC of the *projected* words. These still contain the control broadcasts that projection emits for the `if` and `while` constructs of P; the corresponding source-level MSC is recovered only by erasing them, which is done for the final observable statement. The channel parameter matters only for bookkeeping: in a non-parallel construct it is kept unchanged, while a parallel node replaces it by branch names that are outside the corresponding branch subprograms.

We first state the local prefix property used in the Zipper argument.

Lemma 5.2 (Prefix-freeness of complete local traces)**.** *For every local program S and lifeline A, $\langle\!\langle S \rangle\!\rangle^\dagger$ is prefix-free.*

Proof. By structural induction on S. Atomic programs and ε are immediate. Sequential composition uses the induction hypotheses for the two factors. For conditionals and loops, two complete traces with

one a prefix of the other agree on their first control letter, hence select the same arm or take the same first step (a further iteration versus the loop exit), and the claim follows by the induction hypothesis on the chosen subprogram. For $\mathsf{parloc}(S_1,\dots,S_n)$, suppose two complete tagged traces are comparable by prefix. As the component tags are explicit, restricting both to each component yields complete traces of the corresponding S_j; were the prefix strict, some component trace would be a strict prefix of another complete component trace, contradicting the induction hypothesis. For local programs without parloc, this is ordinary prefix-freeness of $\langle\!\langle S \rangle\!\rangle$. □

By Definition 3.2, every channel name in $C_{o,i} = \{c.o.i\} \cup \mathsf{Ch}^{c.o.i}(P_i)$ begins with the prefix $c.o.i$, so its events lie within branch i of occurrence o (a nested name $c.o.i.o'.i'$ is generated by the inner occurrence o', but still inside branch i of o). Since FIFO matches only equal names, the matching event of a send or receive on such a name lies on the same name, hence again within branch i of o. Thus, in any tuple, the events on the channels $C_{o,i}$ are exactly the branch-i events of o and are matched among themselves. We call this *channel-origin uniqueness*.

Channel-origin uniqueness is a *spatial* separation: it tells parallel occurrences apart—siblings, and successive nodes such as one parloc followed by another—so that a channel name pins down one branch of one occurrence. It does *not* separate distinct dynamic iterations of the *same* occurrence: a parallel occurrence run inside a loop body reuses its branch channels across iterations, since static projection cannot allocate a fresh channel per dynamic iteration. Hence a branch message could a priori be FIFO-matched across iterations, or across the prefix/continuation boundary, the only reuse the occurrence labelling leaves behind. This residual, *temporal* reuse is handled not by channel names but by counting: the following two-sided invariant of the ZIPPERGEN Zipper proof rules it out once one endpoint has completed the subprogram. Both directions are used below: direction (1) for the branch restriction, direction (2) for the replacement of U by $\bar{U}$ in condition 4. For a directed channel $A \to B[d]$ and a local word w, write $\#\mathsf{snd}_{A\to B[d]}(w)$ for the number of send letters on $A \to B[d]$ in w and $\#\mathsf{rcv}_{A\to B[d]}(w)$ for the number of receive letters on $A \to B[d]$ in w.

Lemma 5.3 (FIFO-count invariant)**.** *Let P be a well-typed extended global subprogram reached under channel c. Let $(u_X)_X$ and $(v_X)_X$ be tuples of local words such that, for every lifeline X,*

$$u_X \in \langle\!\langle \pi^c_X(P) \rangle\!\rangle_{\mathsf{pref}} \qquad \text{and} \qquad v_X \neq \varepsilon \implies u_X \in \langle\!\langle \pi^c_X(P) \rangle\!\rangle,$$

and suppose that $(u_X v_X)_X$ is an MSC. Fix a directed channel $A \to B[d]$ occurring in the projection of P, that is, $d = c$ or $d \in \mathsf{Ch}^c(P)$. Then:

(1) *If $u_A \in \langle\!\langle \pi^c_A(P) \rangle\!\rangle$, then*

$$\#\mathsf{rcv}_{A\to B[d]}(u_B) \le \#\mathsf{snd}_{A\to B[d]}(u_A).$$

Consequently, every $A \to B[d]$ receive in u_B is FIFO-matched to a send in u_A.

(2) *If $u_B \in \langle\!\langle \pi^c_B(P) \rangle\!\rangle$, then*

$$\#\mathsf{snd}_{A\to B[d]}(u_A) \le \#\mathsf{rcv}_{A\to B[d]}(u_B).$$

Consequently, every $A \to B[d]$ send in u_A that is matched in $(u_X v_X)_X$ is FIFO-matched to a receive in u_B.

Proof. The proof is by induction on P, proving both inequalities together. We give the argument for (1); the proof of (2) is analogous, with the roles of the completed endpoint and the other endpoint exchanged.

For an atomic message, action, or empty program, the claim follows directly from the local trace semantics and the FIFO definition. In particular, for a message $A \to B$ on channel c, a complete sender-side

trace contains the single corresponding send, while a receiver-side prefix contains at most the matching receive. Atomic programs not using the channel $A \to B[d]$ contribute zero to the relevant counts.

Sequential composition uses the unique prefix decomposition induced by the local trace semantics. The counts on $A \to B[d]$ split into the contributions of the two sequential components, and the induction hypotheses for the two components are summed.

Conditionals and loops are the sequential ZIPPERGEN cases. The projected control broadcasts synchronize all participants with the owner on the selected branch and, for loops, on the number of completed iterations. Once the completed endpoint has finished the local projection of the construct, the other endpoint cannot have consumed more $A \to B[d]$ communications of that construct than the completed endpoint has produced. This is exactly the FIFO-count invariant of the sequential Zipper argument, with the current channel name carried along.

It remains to consider $P = \mathsf{parallel}^o(P_1, \ldots, P_n)$. If $d = c$, then no send or receive of the parloc projection uses d, so both relevant counts are zero. Otherwise, by channel-origin uniqueness, d belongs to a unique branch-channel set

$$C_i = \{c.o.i\} \cup \mathsf{Ch}^{c.o.i}(P_i),$$

so every $A \to B[d]$ letter belongs to branch i.

Choose tagged lifts of the prefixes u_X in the outer parloc projection, taking a *complete* tagged lift whenever u_X is itself complete for that projection. This is possible because the untagged complete-trace language is the forget-image of the tagged one: a complete u_X equals $\mathsf{forget}(u_X^\dagger)$ for some *complete* tagged trace $u_X^\dagger$. (We cannot instead argue that every lift of a complete trace is complete: forget is many-to-one, and the untagged parloc language is not prefix-free, so a complete untagged trace may also have lifts that are strict tagged prefixes.) Let u_X^i be the outer-tag-i component of such a lift, with all tags then forgotten (the FIFO-count statement is about untagged words). Thus $u_X^i \in \langle\!\langle \pi_X^{c.o.i}(P_i) \rangle\!\rangle_{\mathsf{pref}}$, and if u_X is complete then u_X^i is complete for $\pi_X^{c.o.i}(P_i)$, because a complete tagged parloc trace restricts to complete component traces. The choice of tagged lift does not affect the $A \to B[d]$ counts, since communication letters on d are identified by their channel name.

Let v_X^i be the subword of v_X consisting of the send and receive letters on channels in C_i; the continuation may reuse these channels, and such communications are exactly what v_X^i retains. Since FIFO matching is per directed channel, every receive on a channel in C_i is matched, in $(u_X v_X)_X$, to a send on that same channel, hence again in C_i. Deleting all other events preserves payload matching and acyclicity, so $(u_X^i v_X^i)_X$ is an MSC. Also, if $v_X^i \neq \varepsilon$, then $v_X \neq \varepsilon$, so the hypothesis gives that u_X is complete for the outer parloc program, and therefore u_X^i is complete for branch i.

Thus the induction hypothesis applies to P_i under channel $c.o.i$. If u_A is complete for the outer parloc program, then u_A^i is complete for the branch projection, and the induction hypothesis gives

$$\#\mathsf{rcv}_{A \to B[d]}(u_B^i) \leq \#\mathsf{snd}_{A \to B[d]}(u_A^i).$$

These are exactly the corresponding counts in u_B and u_A, because all letters on d belong to branch i. This proves (1); the argument for (2) is the same, using completeness of u_B instead of u_A.

In both directions the matching claim follows from the inequality: a prefix holds the lowest-indexed letters on each directed channel, so a communication whose FIFO index is within the bound has its matching event in the prefix. □

The proof below uses two elementary, mutually inverse facts about the decorated trace semantics of a parloc node. *Restriction:* a tagged prefix restricts componentwise to tagged prefixes of the components,

and a tagged complete trace restricts to tagged complete component traces. *Assembly:* conversely, interleaving complete tagged traces of the components yields a complete tagged trace of the parloc node whose componentwise restrictions are those traces (Definition 4.2). Component tags are proof witnesses and do not add runtime events; channel names, not tags, determine FIFO matching for send and receive letters.

Lemma 5.4 (Zipper Lemma)**.** *Let P be a well-typed extended program, and fix a channel name c. For each lifeline A, let $S_A = \pi^c_A(P)$. Let $U = (u_A)_A$ and $V = (v_A)_A$ be tuples of local words such that:*

- *for every A: $u_A \in \langle\!\langle S_A \rangle\!\rangle_{\mathsf{pref}}$;*
- *for every A: $v_A \neq \varepsilon \implies u_A \in \langle\!\langle S_A \rangle\!\rangle$ (if A has started the continuation, it has finished P);*
- *$(u_A v_A)_A$ is an MSC.*

Then there exists $\bar{U}$ such that $\mathsf{ZipPost}^c_P(U, \bar{U}, V)$.

Proof. We prove, by induction on P, a decorated strengthening of the lemma. For each lifeline A, fix a tagged lift $u^\dagger_A \in \langle\!\langle S_A \rangle\!\rangle^\dagger_{\mathsf{pref}}$ with $\mathsf{forget}(u^\dagger_A) = u_A$, chosen complete whenever $v_A \neq \varepsilon$. This is possible because then u_A is complete, and complete untagged traces are forget-images of complete tagged traces. The strengthened claim is that there exist tagged completions $\bar{u}^\dagger_A \in \langle\!\langle S_A \rangle\!\rangle^\dagger$ such that

$$u^\dagger_A \preceq \bar{u}^\dagger_A \qquad \text{and} \qquad v_A \neq \varepsilon \implies \bar{u}^\dagger_A = u^\dagger_A,$$

and such that, writing $\bar{U} = (\mathsf{forget}(\bar{u}^\dagger_A))_A$, conditions 2–4 of Definition 5.1 hold. Since tags affect neither FIFO matching nor the MSC relation, this strengthened claim implies the stated lemma.

The induction follows [3]. For ε, messages, actions, sequential composition, conditionals, and loops there is no new top-level parloc, so the ordinary construction applies recursively, with the channel name c carried unchanged and the tags untouched. The only case that manipulates tags is $P = \mathsf{parallel}^o(P_1, \ldots, P_n)$, with $S_A = \mathsf{parloc}(S^1_A, \ldots, S^n_A)$ and $S^i_A = \pi^{c.o.i}_A(P_i)$.

Channel discipline. Let $C_i = \{c.o.i\} \cup \mathsf{Ch}^{c.o.i}(P_i)$ be the channels of branch i. For this fixed occurrence the sibling sets $C_1, \ldots, C_n$ are pairwise disjoint, so a send or receive on a channel in C_i belongs to branch i, and its FIFO matching event, if any, lies on a channel in C_i.

Branch prefixes. Restricting $u^\dagger_A$ to its outer-tag-i letters and stripping that tag gives $u^{i,\dagger}_A \in \langle\!\langle S^i_A \rangle\!\rangle^\dagger_{\mathsf{pref}}$, complete whenever $u^\dagger_A$ is; put $u^i_A = \mathsf{forget}(u^{i,\dagger}_A)$ and $U^i = (u^i_A)_A$ (communication letters are sorted into branches by their channel, local actions and owner choices by their outer tag). We show U^i is an MSC. Acyclicity and payload matching are inherited from $(u_A v_A)_A$ by deleting events; only receive-matching needs an argument. Consider a branch-i receive in u^i_B on a channel $d \in C_i$ from A; in $(u_X v_X)_X$ it is matched by a send on d, which lies in u^i_A or in v_A.

Let v^i_X be the subword of v_X on the channels of C_i. Then $(u^i_X v^i_X)_X$ is an MSC: every C_i-receive keeps its corresponding FIFO-matching event and deleting the other events creates no cycle. Moreover, $(u^i_X)_X, (v^i_X)_X$ satisfy the Zipper hypotheses for P_i: in particular $v^i_X \neq \varepsilon$ forces $v_X \neq \varepsilon$, so u_X is complete and, by the choice of lift, u^i_X is complete. (Recall that continuation v^i_X need not be a trace of any program.) Recall the matching send lies in u^i_A or in v_A. If $v_A = \varepsilon$ it lies in u^i_A. Otherwise $v_A \neq \varepsilon$, so u^i_A is a complete trace of S^i_A, and Lemma 5.3(1) for P_i places it in u^i_A. Either way the receive is matched within U^i, so U^i is an MSC.

Branch completions and assembly. Applying the induction hypothesis to each P_i with prefix $(u^{i,\dagger}_A)_A$ and empty continuation yields tagged completions $\bar{u}^{i,\dagger}_A = u^{i,\dagger}_A \delta^i_A \in \langle\!\langle S^i_A \rangle\!\rangle^\dagger$, each $\bar{U}^i = (\mathsf{forget}(\bar{u}^{i,\dagger}_A))_A$ a complete MSC. Append the suffixes $\delta^1_A, \ldots, \delta^n_A$ to $u^\dagger_A$, with outer tags restored, forming $\bar{u}^\dagger_A$; its outer-tag-i

part is the complete trace $\bar{u}_A^{i,\dagger}$, so by the semantics of parloc, $\bar{u}_A^{\dagger} \in \langle\!\langle S_A \rangle\!\rangle^{\dagger}$. Put $\bar{u}_A = \mathsf{forget}(\bar{u}_A^{\dagger})$, so $u_A \preceq \bar{u}_A \in \langle\!\langle S_A \rangle\!\rangle$. If $v_A \neq \varepsilon$ then $u_A^{\dagger}$, hence each $u_A^{i,\dagger}$, is complete; as $u_A^{i,\dagger} \preceq \bar{u}_A^{i,\dagger}$, prefix-freeness (Lemma 5.2) gives $\delta_A^i = \varepsilon$, whence $\bar{u}_A = u_A$.

Condition 2. $\bar{U} = (\bar{u}_A)_A$ is a complete MSC. Completeness and payload matching are inherited from the complete branch MSCs $\bar{U}^i$, the C_i being disjoint. For acyclicity, give the old letters level 0 and the letters of δ_A^i level i. Local order is level-monotone (old letters, then $\delta^1, \ldots, \delta^n$). So are FIFO edges: a match within level 0 or within one δ^i stays there, and an old send completed by a new receive induces an edge from level 0 to level i. No edge decreases the level, since a δ^i-send matches neither an old receive (old receives are already matched to old sends in U^i) nor a δ^j-receive with $j \neq i$ (the C_i are disjoint). A cycle would thus stay within one level. Level 0 is acyclic as a subgraph of $(u_A v_A)_A$, and level i as a subgraph of $\bar{U}^i$.

Conditions 3 and 4. Condition 3 is the hypothesis that $(u_A v_A)_A$ is an MSC. For condition 4 take a receive of B in v_B on $A \to B[d]$. Then $v_B \neq \varepsilon$, so u_B is complete. If $v_A = \varepsilon$, all $A \to B[d]$ sends lie in u_A, so Lemma 5.3(2) gives $\#\mathsf{snd}_{A\to B[d]}(u_A) \leq \#\mathsf{rcv}_{A\to B[d]}(u_B)$. With $(u_A v_A)_A$ an MSC this leaves no $A \to B[d]$ receive for v_B, a contradiction. Hence $v_A \neq \varepsilon$, so $\bar{u}_A = u_A$ and the receive keeps its matching send (payload matching is inherited pairwise). For acyclicity, split the events into old, completion (δ), and continuation (V). No lifeline carries both a δ and a v letter (since $\delta_A \neq \varepsilon$ only when $v_A = \varepsilon$), and both follow the old letters. Each δ receive is matched inside $\bar{U}$, and since $\bar{U}$ is complete, each δ send is matched inside $\bar{U}$ as well. So no FIFO edge joins δ and V. Across the boundary no send matches an old receive either, for two reasons. A completion send cannot, since every old receive is already matched to an old send inside $\bar{U}$. A continuation send cannot, since its sender is complete, so Lemma 5.3(1) already puts the matching event of every old receive within the old prefix. Hence every cross-boundary edge runs from an old letter to a δ or v letter, never the reverse, and none joins δ and V. Any cycle in $(\bar{u}_A v_A)_A$ therefore lies in $\bar{U}$ or in $(u_A v_A)_A$, both acyclic; so $(\bar{u}_A v_A)_A$ is an MSC. Forgetting the tags, $\mathsf{ZipPost}_P^c(U, \bar{U}, V)$ holds. □

Definition 5.5 (Deadlock-free)**.** A distributed program $\mathscr{D}$ is *deadlock-free* if for every $M \in [\![\mathscr{D}]\!]_{\mathsf{pref}}$, there exists $M' \in [\![\mathscr{D}]\!]$ with $M \preceq M'$.

Theorem 5.6 (Deadlock-freedom)**.** *Let P be a well-typed extended* ZIPPERGEN *program with nested* parallel *regions. Then D_P is deadlock-free.*

Proof. Apply Lemma 5.4 with $V = (\varepsilon)_A$. Let $M = (m_A)_A \in [\![D_P]\!]_{\mathsf{pref}}$. Then $m_A \in \langle\!\langle S_A \rangle\!\rangle_{\mathsf{pref}}$ and $(m_A)_A$ is an MSC. The second Zipper hypothesis is vacuous (all $v_A = \varepsilon$), and the final MSC hypothesis is satisfied. The lemma yields $\bar{U}$ with $\mathsf{ZipPost}_P^{\mathsf{main}}(M, \bar{U}, (\varepsilon)_A)$. In particular $\bar{u}_A \in \langle\!\langle S_A \rangle\!\rangle$ and $\bar{U}$ is a complete MSC, so $\bar{U} \in [\![D_P]\!]$, with $M \preceq \bar{U}$. □

The erasure in the theorem below is only control erasure: it removes the compiler-inserted control broadcasts, but it does not remove channel names. Thus both the projected semantics and the source-level observation are compared as MSCs over the same channel-named alphabet. These channel names are part of the semantic representation of parallel regions, since they keep FIFO matching for different branches separate.

Theorem 5.7 (Inductive Correctness)**.** *Let P be a well-typed extended program and D_P its projected distributed program. Then the following hold:*

(1) *For every $M \in [\![P]\!]_{\mathsf{src}}$, there exists $\widehat{M} \in [\![D_P]\!]$ such that $\mathsf{erase}_{\mathsf{ctrl}}(\widehat{M}) = M$.*

(2) *For every $\widehat{M} \in [\![D_P]\!]$, $\mathsf{erase}_{\mathsf{ctrl}}(\widehat{M}) \in [\![P]\!]_{\mathsf{src}}$.*

Proof. We prove the stronger channel-parametric statement $[\![D_P^c]\!] = \mathsf{Enr}^c(P)$ for every channel name c, by induction on P.

The cases ε, `act`, `msg`, sequential composition, `if`, and `while` are exactly those of [3]: no non-parallel construct alters the current channel, so the base proof applies with c carried unchanged—in particular used throughout the decision blocks of `if` and `while`, whose reserved control tags $\mathsf{tag}^Q_{\mathsf{ctrl}}$ keep the broadcasts separate from ordinary messages. (The sequential case is the Zipper argument of Lemma 5.4 together with cancellation of MSC concatenation.) The only new case is $P = \mathsf{parallel}^o(P_1, \ldots, P_n)$.

It remains to consider $P = \mathsf{parallel}^o(P_1, \ldots, P_n)$. If $M \in \mathsf{Enr}^c(P)$, then by definition M is a complete MSC shuffle of some $M_i \in \mathsf{Enr}^{c.o.i}(P_i)$. By induction, $M_i \in [\![D_{P_i}^{c.o.i}]\!]$ for every i. On each lifeline, the same shuffle yields a word in $\langle\!\langle \mathsf{parloc}(\pi_A^{c.o.1}(P_1), \ldots, \pi_A^{c.o.n}(P_n)) \rangle\!\rangle$ by Definition 4.2. Hence $M \in [\![D_P^c]\!]$.

Conversely, let $M \in [\![D_P^c]\!]$. Each lifeline component of M is a complete trace of the outer parloc projection, so it has a complete tagged lift; choose such lifts, restrict them to component i, and forget all tags, obtaining untagged tuples M_i. By channel-origin uniqueness the sibling branch channel sets $C_1, \ldots, C_n$ are disjoint, so send and receive letters restrict component by component. Since M is a complete MSC and the chosen lifts are complete, receive matching, payload matching, completeness, and acyclicity are inherited by these restrictions. The branch-numbering tag on a local action or owner-choice letter does not affect matching, since such letters are single-lifeline events. Thus $M_i \in [\![D_{P_i}^{c.o.i}]\!]$. By induction, $M_i \in \mathsf{Enr}^{c.o.i}(P_i)$. The original tuple M is a complete MSC shuffle of the M_i, so $M \in \mathsf{Enr}^c(\mathsf{parallel}^o(P_1, \ldots, P_n))$ by Definition 3.4.

Taking $c = \mathsf{main}$ gives the internal equality $[\![D_P]\!] = \mathsf{Enr}^{\mathsf{main}}(P)$. The two erasure statements follow directly from the definition $[\![P]\!]_{\mathsf{src}} = \mathsf{erase}_{\mathsf{ctrl}}(\mathsf{Enr}^{\mathsf{main}}(P))$. □

For programs without parallel, all user messages use the initial channel name, and forgetting this uniform name recovers the channel-free statement of [3].

6 Conclusion

We have presented structured parallel regions for projected workflows. The construct lets a source program run full workflow branches in parallel, including branches that share lifelines. Projection remains syntax-directed: each lifeline receives a local parloc program over the branch projections, with ε components for branches in which it does not participate.

The proof relies on three mechanisms. Branch-local channel names separate FIFO matching across branches. The internal enriched semantics filters enriched branch shuffles to those that are complete MSCs, while still guaranteeing existence by serializing branches. The channel-parametric Zipper lemma then shows that every MSC prefix of the projected distributed program extends to a complete projected MSC. This extends the deadlock-freedom and semantic correctness results of ZIPPERGEN to nested parallel regions. Static criteria ensuring that the semantic filter is redundant are a natural direction for future work.

References

[1] Rajeev Alur, Kousha Etessami & Mihalis Yannakakis (2005): *Realizability and verification of MSC graphs*. *Theor. Comput. Sci.* 331(1), pp. 97–114, doi:10.1016/J.TCS.2004.09.034. Available at https://doi.org/10.1016/j.tcs.2004.09.034.

[2] Rajeev Alur & Mihalis Yannakakis (1999): *Model Checking of Message Sequence Charts*. In Jos C. M. Baeten & Sjouke Mauw, editors: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 114–129, doi:10.1007/3-540-48320-9_10. Available at https://doi.org/10.1007/3-540-48320-9_10.

[3] Benedikt Bollig, Matthias Függer & Thomas Nowak (2026): *Provable Coordination for LLM Agents via Message Sequence Charts*. *CoRR* abs/2604.17612, doi:10.48550/arXiv.2604.17612. arXiv:2604.17612. To appear in ISoLA 2026.

[4] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, ACM, pp. 263–274, doi:10.1145/2429069.2429101. Available at https://doi.org/10.1145/2429069.2429101.

[5] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Session*. *Log. Methods Comput. Sci.* 8(1), doi:10.2168/LMCS-8(1:24)2012. Available at https://doi.org/10.2168/LMCS-8(1:24)2012.

[6] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida & Luca Padovani (2016): *Global progress for dynamically interleaved multiparty sessions*. *Math. Struct. Comput. Sci.* 26(2), pp. 238–302, doi:10.1017/S0960129514000188. Available at https://doi.org/10.1017/S0960129514000188.

[7] Philippe Darondeau, Blaise Genest & Loïc Hélouët (2008): *Products of Message Sequence Charts*. In Roberto M. Amadio, editor: *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 458–473, doi:10.1007/978-3-540-78499-9_32. Available at https://doi.org/10.1007/978-3-540-78499-9_32.

[8] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In Helmut Seidl, editor: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2_10. Available at https://doi.org/10.1007/978-3-642-28869-2_10.

[9] Thomas Gazagnaire, Blaise Genest, Loïc Hélouët, P. S. Thiagarajan & Shaofa Yang (2009): *Causal Message Sequence Charts*. *Theor. Comput. Sci.* 410(41), pp. 4094–4110, doi:10.1016/J.TCS.2009.06.013. Available at https://doi.org/10.1016/j.tcs.2009.06.013.

[10] Blaise Genest, Anca Muscholl, Helmut Seidl & Marc Zeitoun (2006): *Infinite-state high-level MSCs: Model-checking and realizability*. *J. Comput. Syst. Sci.* 72(4), pp. 617–647, doi:10.1016/J.JCSS.2005.09.007. Available at https://doi.org/10.1016/j.jcss.2005.09.007.

[11] Lorenzo Gheri & Nobuko Yoshida (2023): *Hybrid Multiparty Session Types: Compositionality for Protocol Specification through Endpoint Projection*. *Proc. ACM Program. Lang.* 7(OOPSLA1), pp. 112–142, doi:10.1145/3586031. Available at https://doi.org/10.1145/3586031.

[12] Loïc Hélouët, Thibaut Hénin & Christophe Chevrier (2006): *Automating Scenario Merging*. In Reinhard Gotzhein & Rick Reed, editors: *System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31 - June 2, 2006, Revised Selected Papers*, Lecture Notes in Computer Science, Springer, pp. 64–81, doi:10.1007/11951148_5. Available at https://doi.org/10.1007/11951148_5.

[13] Andrew K. Hirsch & Deepak Garg (2022): *Pirouette: higher-order typed functional choreographies*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–27, doi:10.1145/3498684. Available at https://doi.org/10.1145/3498684.

[14] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695. Available at https://doi.org/10.1145/2827695.

[15] ITU-T (2011): *Recommendation Z.120: Message Sequence Chart (MSC)*. Technical Report, International Telecommunication Union.

[16] Jules Jacobs, Stephanie Balzer & Robbert Krebbers (2022): *Connectivity graphs: a method for proving deadlock freedom based on separation logic*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–33, doi:10.1145/3498662. Available at https://doi.org/10.1145/3498662.

[17] Jules Jacobs, Stephanie Balzer & Robbert Krebbers (2022): *Multiparty GV: functional multiparty session types with certified deadlock freedom*. *Proc. ACM Program. Lang.* 6(ICFP), pp. 466–495, doi:10.1145/3547638. Available at https://doi.org/10.1145/3547638.

[18] Joost-Pieter Katoen & Lennard Lambert (1998): *Pomsets for message sequence charts*. In Hartmut König & Peter Langendörfer, editors: *Formale Beschreibungstechniken für verteilte Systeme, 8. GI/ITG-Fachgespräch, Cottbus, 4. und 5. Juni 1998*, Verlag Shaker, pp. 197–207.

[19] Jacques Klein, Benoît Caillaud & Loïc Hélouët (2004): *Merging Scenarios*. In Juan Bicarregui, Andrew Butterfield & Alvaro Arenas, editors: *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2004, Linz, Austria, September 20-21, 2004*, Electronic Notes in Theoretical Computer Science, Elsevier, pp. 193–215, doi:10.1016/J.ENTCS.2004.08.065. Available at https://doi.org/10.1016/j.entcs.2004.08.065.

[20] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In Christel Baier & Holger Hermanns, editors: *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 233–247, doi:10.1007/11817949_16. Available at https://doi.org/10.1007/11817949_16.

[21] Rupak Majumdar, Madhavan Mukund, Felix Stutz & Damien Zufferey (2021): *Generalising Projection in Asynchronous Multiparty Session Types*. In Serge Haddad & Daniele Varacca, editors: *32nd International Conference on Concurrency Theory, CONCUR 2021, Virtual Conference, August 24-27, 2021*, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:24, doi:10.4230/LIPICS.CONCUR.2021.35. Available at https://doi.org/10.4230/LIPIcs.CONCUR.2021.35.

[22] Fabrizio Montesi (2013): *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen.

[23] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In Pedro R. D'Argenio & Hernán C. Melgratti, editors: *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30. Available at https://doi.org/10.1007/978-3-642-40184-8_30.

[24] Gan Shen, Shun Kashiwa & Lindsey Kuper (2023): *HasChor: Functional Choreographic Programming for All (Functional Pearl)*. *Proc. ACM Program. Lang.* 7(ICFP), pp. 541–565, doi:10.1145/3607849. Available at https://doi.org/10.1145/3607849.