Event Structure Semantics of Orc

Sidney Rosario[1], David Kitchin[3], Albert Benveniste[1], William Cook[3], Stefan Haar[4], and Claude Jard[2]

[1] Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France
[2] Irisa/ENS Cachan, Campus de Beaulieu, 35042 Rennes cedex, France
[3] The University of Texas at Austin, Department of Computer Sciences, Austin, USA
[4] Irisa/Inria Rennes and SITE, University of Ottawa, Canada

Abstract. One challenge in developing wide-area distributed applications is analyzing the system's non-functional properties, including timing constraints and internal dependencies that can affect quality of service. Analysis of non-functional properties requires a precise formal semantics for the language in which the system is written; but labelled transition systems and trace semantics, which are commonly used for this purpose, do not facilitate this kind of analysis. Event structures provide an explicit representation of the the causal dependencies between events in the execution of a system. But event structures are difficult to construct compositionally, because they cannot easily represent fragments of a computation. In this paper we present a partial-order semantics based on *heaps* (an explicitly encoded form of occurrence nets with read arcs), which naturally represent fragments of behavior. Heaps are then easily translated into asymmetric event structures. The semantics is developed for Orc, an orchestration language in which concurrent services are invoked to achieve a goal while managing time-outs, exceptions, and priority. Orc, and this new semantics, are being used to study quality of service (QoS) for wide area orchestrations.

1 Introduction

Orc is a structured language for computation orchestration, in which concurrent services are invoked to achieve a goal while managing time-outs, exceptions, and priority [12]. The operational semantics of Orc was first defined as a labeled transition system. A denotational semantics of Orc has also been defined; the denotations are sets of traces, which explicitly represent the observable behavior of an Orc program [7]. For other studies of Orc semantics see [3], where the authors link the Orc language to Petri nets and the join calculus, and [14], where Orc expressions are translated to colored Petri net systems. On the other hand, a number of papers have been devoted to the semantics of the most widely used language for orchestration, namely BPEL, see [6,9,13,15,8] and the tutorial [16]. Still, very little has been done toward getting, for orchestration languages, a semantics that is suitable for Quality of Service (QoS) studies.

Analyzing QoS or non-functional properties, like timing constraints derived from the critical path of dependencies, can be quite difficult with either an operational or a denotational trace semantics. The problem is that neither of these

semantics exhibits the causality constraints that govern concurrent execution. These causality constraints can be represented explicitly as partial orders over events. With a partial order semantics, analysis and verification of programs are facilitated, and translations between different formalisms can be checked for correctness. Last but not least, partial order representations are crucial for evaluating overall durations of programs: time-consuming actions that run in parallel increase the overall delay less than actions that have to occur sequentially; see [11,10] for more on this type of dynamics. The partial order semantics is therefore crucial for the QoS analyses for orchestrated services [14]. In this paper we develop a partial order semantics of Orc in terms of *asymmetric event structures* [1]. An event structure is a set of events with one or more relations that constrain the allowed sequences of events. Asymmetric event structures have an *asymmetric conflict* relation, $a \nearrow b$, which states that event b cannot precede event a in a same execution. Asymmetric conflict is convenient to express preemption or termination, which is an essential feature needed for wide area computing and offered by Orc. In Orc, an execution A can be preempted at the instant when a particular event e occurs. The preemption of A by e is expressed by imposing $a \nearrow e$ for all events a in A, which asserts that no event in A can occur after e. In other words, e terminates the execution A. The asymmetric event structures for an Orc expression is defined by two steps.

The first step is a compositional translation of Orc expressions into mathematical structures called *heaps*, introduced in Section 2.2. Heaps are sets of inductively defined events, following a method originally proposed by Esparza [5] to encode net unfoldings. Heaps are useful for two reasons. First, they provide a concrete representation of asymmetric event structures that is suitable for effective coding of algorithms in software. Second, and more importantly, they can specify fragments of computations that refer to virtual events offered by an execution from another heap. The latter feature proved extremely useful for deriving the heap semantics of Orc, structurally.

In the second step, the heap is converted into an asymmetric event structure which is a recognized semantic domain, equipped with well defined notions of *configurations* to model partially ordered executions. A correspondence of these asymmetric event structures with the existing sequential trace semantics of Orc is also shown.

2 Asymmetric Event Structures and Heaps

In this section we recall the needed background on Asymmetric Event Structures (AES). Then we motivate the need for the new concept of *heap* and introduce it. Finally, we show how to generate AES from heaps.

2.1 Asymmetric Event Structures with Labels

Following [17,1], an *Asymmetric Event Structure* (AES) is a model of computation consisting of a set of events and two associated binary relations, the *causality*

relation $\preceq$ and the *asymmetric conflict* relation $\nearrow$. If for events e and e', $e \preceq e'$ holds, then e must occur before e' can occur. If $e \nearrow e'$ holds, then the occurrence of e' preempts the occurrence of e in the future. Thus if both e and e' occur in an execution, e necessarily happens before e'. In this sense, $\nearrow$ can also be seen as a "weak causality" relation.

Formally, an AES is a tuple $\mathbf{G} = (E, \preceq, \nearrow)$, where E is a set of *events*, and $\preceq$ and $\nearrow$ are the *causality* and *asymmetric conflict* binary relations over E, satisfying the following conditions:

1. $\preceq$ is a partial order, and $\lfloor e \rfloor =_{\text{def}} \{e' \in E \mid e' \preceq e\}$ is finite;
2. $\forall e, e' \in E$:

$$e \prec e' \Rightarrow e \nearrow e' \tag{1}$$

$$\text{the restriction of } \nearrow \text{ to } \lfloor e \rfloor \text{ is acyclic} \tag{2}$$

$$e \#^a e' \Rightarrow e \nearrow e' \tag{3}$$

where $\#^a$ is the *conflict relation*, which relates events that preempt each other. For two events, if $e \nearrow e'$ and $e' \nearrow e$ then $e \#^a e'$, and only one of e and e' can occur in an execution. The conflict relation finds sets of mutually conflicting events using this recursive definition:

$$e_0 \nearrow e_1 \nearrow \ldots e_n \nearrow e_0 \Rightarrow \#^a(e_0, \ldots, e_n) \tag{4}$$

$$[\#^a(A \cup \{e\})] \wedge [e \preceq e'] \Rightarrow \#^a(A \cup \{e'\}) \tag{5}$$

The second condition ensures that a conflict with e is inherited by all the events caused by e.

Given an event structure, a *configuration* is a set of events that obey the causality and conflict constraints, and so represent a valid execution instance of the event structure. For $\mathbf{G} = (E, \preceq, \nearrow)$ an AES, a configuration of $\mathbf{G}$ is a set $\kappa \subseteq E$ of events such that

1. the restriction of $\nearrow$ to κ is well-founded;
2. $\{e' \in \kappa \mid e' \nearrow e\}$ is finite for every $e \in \kappa$;
3. κ is left-closed with respect to $\preceq$, i.e., $\forall e \in \kappa, e' \in E$, $e' \preceq e$ implies $e' \in \kappa$.

For our coding of Orc, we will need to label the events. Thus we shall consider *Labeled AES* (LAES), which are tuples of the form $\mathbf{G} = (E, \preceq, \nearrow, \lambda)$, where $\lambda : E \mapsto \Lambda$, ($\Lambda$ is a set of labels) is the labeling (partial) function.

Discussion: from event structures to heaps. Asymmetric event structures allow an event to occur only if its causes have already occurred, and it is not prevented by the occurrence of some other event. This yields a simple and elegant mathematical model for complete concurrent systems that, in all its variants, comes equipped with a comprehensive categorical apparatus [1].

Although event structures work well for complete programs, they cannot easily represent fragments of behavior. Such fragments arise naturally when constructing the behavior of a program from the behaviors of the subexpressions in

the program – as is the standard practice in denotational semantics. For such formalisms, structural translation of programs to (asymmetric) event structures cannot be directly achieved.

By offering the additional concept of *place*, Petri nets and their extensions and variants [17,1] make structural translation easier. Explicit encoding of places allows one fragment to depend upon resources supplied by another fragment. Other features of wide area languages are not so easily supported by Petri nets; modelling dynamic creation of processes requires non-trivial extensions of nets, such as, e.g., net systems [2]. These extensions require another layer of semantics to specify their executions. Therefore, using such Petri net extensions results in a complex two-stage semantics: from the formalism to, e.g., net systems, and, from net systems to their semantic domain. Such a translation was proposed in [14] for Orc, resulting in excessive formalism and complex software coding.

So, a natural idea consists in bypassing the above two-stage approach, by considering directly occurrence nets, with *read arcs*. To be more effective and get close to implementation, we decided in addition to use an explicit inductive coding of such occurrence nets, following the technique first proposed by Esparza et al. [5]. This results in the notion of *heap* described in the next section. The subclass of "effective" heaps translate immediately into asymmetric event structures.

2.2 Heaps

Heaps are sets of events coded in a particular form. A heap event is encoded based on the *conditions* that enable its occurrence. The enabling condition can either be *consumed* by the event or can be *read* and not consumed. The conditions in turn, refer to the events that created them. More precisely:

$$\begin{aligned} \text{event} &= (\text{ consume conditions, read conditions, label }) \\ \text{condition} &= (\text{ cause event, mark }) \end{aligned} \tag{6}$$

where

- *consume conditions* is the set of conditions that are consumed by the event;
- *read conditions* is the set of conditions that are only read (and not consumed) by the event;
- *label* is a label (for our use in Orc semantics, it will be the Orc action performed by the event);
- *mark* is a label to distinguish different conditions created by an event.

We formalize this next.

Definition 1. *Call* heap *a tuple* (E, C, S, A, M)*, where:*

1. *E and S are two sets of* events *such that $E \subseteq S$, C is a set of* conditions*, A is an alphabet of* labels*, and M is a set of* marks*.*

2. *Events $e \in E$ have the following form:*

$$e = ({}^\bullet e, \underline{e}, a) \tag{7}$$

where ${}^\bullet e \subseteq C$ and $\underline{e} \subseteq C$ are the sets of conditions consumed *and* read *by e, respectively, and $a \in A$ is the* label *of e. We require that ${}^\bullet e \cap \underline{e} = \emptyset$ and ${}^\bullet e \cup \underline{e} \neq \emptyset$.*

3. *Conditions $c \in C$ have the following form:*

$$c = (f, \mu) \tag{8}$$

where $f \in S$ and $\mu \in M$ is the mark of condition c.

4. *C and S are minimal, for set inclusion, having the above properties. S is called the* support *of E and C is its set of conditions.*

By abuse of notation, we call E alone a heap, and C_E will denote the set of conditions associated to E. Throughout this paper, we distinguish a fixed event

$$\bot = (\emptyset, \emptyset, \star)$$

called the *dummy event*, where label $\star$ means the absence of label. Note that $\bot$ cannot belong to a heap, it can, however, belong to the support of a heap. Set $E_\bot = E \cup \{\bot\}$. For an event e of the form (7), the set of conditions

$${}^\bullet\underline{e} = {}^\bullet e \cup \underline{e}$$

is called the *pre-set* of e. We define the set of minimal conditions of a heap E, $\mathrm{minConds}(E)$ to be the set

$$\mathrm{minConds}(E) \;=_{\mathrm{def}}\; \{(f, \text{-}) \mid \; (f, \text{-}) \in C_E,\; f \notin E\}$$

Figure 2 shows some example heaps (for Orc expressions). The events of the heap are shown in rectangles, labelled by their corresponding Orc actions. The conditions are the circles. An event has input directed arcs from conditions consumed by it, and undirected dashed arcs from those that are read. Outgoing arcs from an event point to conditions that refer to that event. Minimal conditions refer to the $\bot$ event, which is not shown. A dashed triangle on top of a minimal condition indicates the label of an external event that the condition depends upon. Examples of external events, which are included in the support of the heap, are e, f_1, and f_2.

The conflict and read conditions within the events of a heap define constraints between events, in the style of an event structure. Given a heap E we define the following relations between events in E (superscript * denotes transitive closure):

$$\preceq_E = \lhd^* \text{ where } \lhd = \{(f, e) \mid f^\bullet \cap {}^\bullet\underline{e} \neq \emptyset\} \;\cup\; I_E \tag{9}$$

$$I_E \text{ is the identity relation on } E \times E$$

$$\nearrow'_E = \prec_E \;\cup \left\{ (f, e) \;\middle|\; \exists e' \in E_\bot, e_1 : \begin{bmatrix} (e', \text{-}) \in \; {}^\bullet\underline{f} \;\cap {}^\bullet e_1 \\ \wedge\; e_1 \preceq_E e \end{bmatrix} \right\}$$

$$\nearrow_E = \nearrow'_E \;\cup \{(e, f) \mid e \#^a_E f\} \tag{10}$$

where event variables e, e_1 and f range over E, and the symmetric conflict relation $\#^a_E$ is deduced from $\nearrow'_E$ via (4,5). The reason for the two-step definition of $\nearrow_E$ is that $\nearrow'_E$ satisfies conditions (1,2,4,5), but not necessarily (3). The latter is enforced by second step in the definition, from $\nearrow'_E$ to $\nearrow_E$. Next, equip E with a labeling map

$$\alpha_E(e) =_{\text{def}} a \tag{11}$$

where event $e = ({}^\bullet e, \underline{e}, a)$. We shall denote by

$$\min(E) = \{e \in E \mid \forall f \in E : f \preceq_E e \Rightarrow f = e\} \tag{12}$$

the set of events $e \in E$ that are minimal for the relation $\preceq_E$. For readability, we omit the subscript $_E$ in the sequel. In the SEND heap in Figure 2, $e \preceq f_1$ holds, where e is the event labelled M_{k1} or $k1?v_1$. Also $e \nearrow f_1$ holds for all events e in the heap (except f_1).

Definition 2. *A* configuration *of a heap E is any finite subset κ of E with the following properties:*

1. *the restriction of $\nearrow$ to κ is well-founded;*
2. *$\{e' \in \kappa \mid e' \nearrow e\}$ is finite for every $e \in \kappa$;*
3. *κ is left-closed with respect to $\preceq$, i.e., $\forall e \in \kappa, e' \in E$, $e' \preceq e$ implies $e' \in \kappa$;*
4. *for each event e belonging to κ, if $f^\bullet \cap {}^\bullet\underline{e} \neq \emptyset$ then $f \in E_\perp$.*

As for AES, heap configurations represent legal executions. By condition 3, condition 4 is equivalent to requiring that $f \in \kappa$. Conditions 1–3 coincide with those involved in the definition of configurations for AES, see Section 2.1. Condition 4 is new. For *e.g*, in the SEND heap of Figure 2, any configuration having event f_1 has to include its causal predecessors, *i.e* the events labelled M_{k1} and $k1?v_1$. Event f_2 cannot appear in such a configuration since it is in mutual conflict with f_1, thus Condition 1 would be violated.

Let $\mathbf{Configs}(E)$ be the set of all configurations of heap E.

2.3 From Heaps to LAES

One may expect $(E, \preceq, \nearrow, \alpha)$ to be an LAES. This is not true in general, as certain axioms may be violated (*e.g*, the causal relation $\preceq$ may not be anti-symmetric, or some events may need external events for their enabling). In this section we show how to extract from any heap E, an *effective heap* which has a direct correspondence with an LAES.

Definition 3. *Given a heap E, its effective heap $\mathcal{G}[E]$ is defined as:*

$$\mathcal{G}[E] =_{\text{def}} \bigcup_{\kappa \in \mathbf{Configs}(E)} \kappa.$$

$\mathcal{G}[E]$ possesses a subset of E as its set of events. Generation of $\mathcal{G}[E]$ from a heap E is by pruning and by Definition 2. This generation is constructive. The introduction of effective heap $\mathcal{G}[E]$ is justified by the following result, where symbols $\preceq, \nearrow$, and α are the restrictions, to $\mathcal{G}[E]$, of the relations and map defined in (9), (10), and (11), respectively.

Theorem 1. *$\mathcal{A}[E] = (\mathcal{G}[E], \preceq, \nearrow, \alpha)$ is an LAES. Furthermore, $\mathcal{G}[E]$ is the maximal subset of events of E that induces an LAES.*

Proof Outline. The complete proof is given in Appendix A. The first part is proved by using (9), (10) and Definition 2 to show that relations $\preceq, \nearrow$ on $\mathcal{G}[E]$ satisfy the conditions required for a LAES. The second part is proved by showing that any configuration of a maximal LAES induced by E is contained in $\mathbf{Configs}(E)$ and thus in $\mathcal{G}[E]$.

Remark: The reader should not confuse between the notion of heaps given here and those in [11,10], where the authors study *heaps* formed by blocks representing *durations* of executions in transition systems. Since their heaps are downward causally closed conflict free partial orders, they correspond to *configurations* in our setting, rather than the heaps in the above sense.

2.4 Generic Operations on Heaps

We list here a few operations on heaps that are useful for wide area computing. From now on, *we specialize marks to being lists*, with the usual operations.

- *Marking:* Marking creates distinct copies of a heap. For a heap E and m a mark, E^m is the heap where symbol m has been appended to the mark $\mu(c)$ of each condition $c \in \text{minConds}(E)$. The recursive definitions of events and conditions in E ensures that this operation creates a new instance of E.
- *Disjoint Union:* The disjoint union of heaps E and F where *left* and *right* are fixed marks is:

$$E \uplus F =_{\text{def}} E^{left} \cup F^{right}$$

- *Preemption:* For a heap E and $F \subseteq E$, the preemption of E by F terminates execution of E when any event in F occurs. Formally, $\text{STOP}_F(E)$ is the heap obtained by replacing each event $e = ({}^\bullet e, \underline{e}, a)$ of E by the following event $\varphi(e)$:

$$\varphi(e) =_{\text{def}} \begin{cases} ({}^\bullet e \cup \{(\bot, stop\}, \underline{e}, a) & \text{if } e \in F. \\ ({}^\bullet e, \underline{e} \cup \{(\bot, stop)\}, a) & \text{if } e \notin F. \end{cases} \tag{13}$$

- *Copy:* For two heaps E and F, we define $\text{COPY}_l(E, F)$ to be a copy of E with respect to *context heap* F. For a mark l, $\text{COPY}_l(E, F)$ is a fresh heap obtained by changing all minimal conditions $(e, \mu) \in \text{minConds}(E)$ as follows:

$$(e, \mu) = \begin{cases} (e, (\mu, l)) & \text{if } (e, \mu) \notin C_F \\ (e, \mu) & \text{if } (e, \mu) \in C_F \end{cases} \tag{14}$$

  where C_F is the set of associated conditions of the context heap F. Intuitively, events in E may share conditions (and thus are related) with events in the context heap F. The copy of E with respect to context F keeps these conditions intact in the copy to preserve the relations between the copied events and those in F.

3 Orc Syntax and Semantics

The reader is referred to [12] for an introduction to and motivation for Orc, as a language for wide area computing. The syntax and operational semantics of Orc in the form of SOS rules [7], are given in Figure 1.

$$
\begin{array}{rcll}
f, g, h & \in & \textit{Expression} & ::= M(p) \mid E(p) \mid f \mid g \mid f >x> g \mid f \textbf{ where } x :\in g \mid ?k \\
p & \in & \textit{Actual} & ::= x \mid v \\
 & & \textit{Definition} & ::= E(x) \ \triangleq \ f
\end{array}
$$

$$\frac{k \text{ fresh}}{M(v) \xrightarrow{M_k(v)} ?k} \ (\textsc{SiteCall}) \qquad \frac{f \xrightarrow{a} f' \qquad a \neq !v}{f >x> g \xrightarrow{a} f' >x> g} \ (\textsc{Seq1N})$$

$$?k \xrightarrow{k?v} let(v) \ (\textsc{SiteRet}) \qquad \frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{\tau} (f' >x> g) \mid [v/x].g} \ (\textsc{Seq1V})$$

$$let(v) \xrightarrow{!v} 0 \ (\textsc{Let}) \qquad \frac{f \xrightarrow{a} f'}{f \textbf{ where } x :\in g \xrightarrow{a} f' \textbf{ where } x :\in g} \ (\textsc{Asym1N})$$

$$\frac{f \xrightarrow{a} f'}{f \mid g \xrightarrow{a} f' \mid g} \ (\textsc{Sym1}) \qquad \frac{g \xrightarrow{!v} g'}{f \textbf{ where } x :\in g \xrightarrow{\tau} [v/x].f} \ (\textsc{Asym1V})$$

$$\frac{g \xrightarrow{a} g'}{f \mid g \xrightarrow{a} f \mid g'} \ (\textsc{Sym2}) \qquad \frac{g \xrightarrow{a} g' \qquad a \neq !v}{f \textbf{ where } x :\in g \xrightarrow{a} f \textbf{ where } x :\in g'} \ (\textsc{Asym2})$$

$$\frac{[\![E(x) \ \triangleq \ f \,]\!] \in D}{E(p) \xrightarrow{\tau} [p/x].f} \ (\textsc{Def})$$

Fig. 1. The Syntax (top) Operational Semantics (bottom) of Orc

An Orc expression f can perform action a and transform itself into the expression f', which is denoted by the transition $f \xrightarrow{a} f'$. The actions A and values V are described by the following grammar:

$$
\begin{aligned}
a \in A &::= M_k(v) \mid k?v \mid !v \mid \tau \mid \tau_v \\
v \in V &::= x \mid v_k \mid \mathbf{v}
\end{aligned}
$$

The actions A are the transition labels of the Orc operational semantics, except for the τ_v action which is an intermediary action needed for creating heaps. The x are variable names. They are placeholders for the value which will eventually replace that variable in the expression. The return values v_k are indexed by call handles. They are placeholders for the values returned from site calls. The ground values $\mathbf{v}$ are the constant values which are always available.

Observe the following. Due to rule (DEF), recursive definitions are possible in Orc. Also, rule (ASYM1V) exhibits termination of g upon its first publication.

To simplify the translation, we assume that the Orc programs we consider have distinct variable names. This restriction does not reduce the program's expressivity and can be enforced by a simple syntactic pre-processing step.

4 Denotations for Orc Expressions

In this section, we show how to construct the heap of an Orc program, and then its LAES. We begin with further useful operations on heaps that are specific to Orc. Then, we provide the heap semantics of Orc base expressions and operators.

- *Free Variables:* $E(x)$ is the set of all events in heap E which depend on x.

$$E(x) = \{e \mid e' \preceq_E e, \alpha(e') \in \{M_k(x), !x, \tau_x\}\}$$

  Call x a *free variable* of E if $E(x)$ is nonempty. Let $E(\overline{x})$ be the events in E that do not depend on x: $E(\overline{x}) = E - E(x)$.
- *Publication events:* $!E$ is the set of publication events of heap E:

$$!E = \{e \mid \alpha(e) = !v\}$$

- *Preemption:* Stopping E after the first value publication is defined as:

$$\text{STOP}(E) =_{\text{def}} \text{STOP}_{!E}(E)$$

- *Send:* For a publication event $e = ({}^\bullet e, \underline{e}, !v)$, define the $\tau(e)$ to be the event obtained by changing the label of e as follows:

$$\alpha(e) = \begin{cases} \tau_x \text{ if } \alpha(e) = !x, \text{ for any variable } x \\ \tau \quad \text{otherwise} \end{cases} \tag{15}$$

  The heap $\text{SEND}(E)$ is the heap E where all the publication events e in E are replaced by $\tau(e)$. The publication events are still identifiable by their marks.
- *Link:* For a heap E, a *context heap* C, an event f not belonging to E, and a value v,

$$\text{LINK}(f, v, x, E, C)$$

  is a heap in which variable x is bound to value v after external event f. The *context* heap C identifies parts E that are not affected by the variable binding. $\text{LINK}(f, v, x, E, C)$ is the heap resulting from the following operations:
  1. Create $E' = \text{COPY}_f(E, C)$ a new copy of E with respect to context heap C and marked with label f. In making this copy, each event $e \in E$ has a unique corresponding event $e' = \varphi_f(e) \in E'$.
  2. Change all $e' = ({}^\bullet e', \underline{e}', a) \in E'$ as below, where $e = \varphi_f^{-1}(e')$:

$$e' = \begin{cases} ({}^\bullet e' \cup \{(f, e)\}, \underline{e}', [v/x]a) & \text{if } e' \in \min(E') \\ ({}^\bullet e', \underline{e}', [v/x]a) & \text{if } e' \notin \min(E') \end{cases} \tag{16}$$

The substitution $[v/x]a$ replaces the variable x by v in the action a. If the variable x does not occur in a, the substitution leaves a unchanged. In the heap constructed here, the event f referred by $e' \in \min(E')$ is not in the heap.

– *Receive:* We next construct a heap that can receive any values that is published by another heap. If e is a publication event, $\tau(e)$ is the event e with its action changed according to (15). We define

$$\textsc{recv}_x(E, F, C) = \bigcup_{f \in\, !E, \alpha(f) = !v} \textsc{link}(\tau(f), v, x, F, C)$$

Observe that, if $!E$ is empty, this yields $\textsc{recv}_x(E, F, C) = \emptyset$.

– *Pipe:* The pipe operator allows G to receive publications from F, subject to a context C that identifies parts of G not affected by the communication.

$$\textsc{pipe}_x(F, G, C) = \textsc{send}(F) \ \cup \ \textsc{recv}_x(F, G, C)$$

4.1 Heaps of Base Expressions

For an Orc expression f, $[f]$ is its heap denotation. In the following, *nil* is a distinguished symbol indicating the absence of mark.

$$\begin{aligned}
[\mathbf{0}] &= \emptyset \\
[let(v)] &= \{\, (\{c\}, \emptyset, !v) \,\} \\
&\quad \text{where condition } c = (\bot, nil) \\
[?k] &= \{\, e = (\{c_1\}, \emptyset, k?v_k), (\{c_2\}, \emptyset, !v_k) \,\} \\
&\quad \text{where condition } c_1 = (\bot, nil),\ c_2 = (e, nil) \\
[M(v)] &= \{\, e = (\{c_1\}, \emptyset, M_k(v)), f = (\{c_2\}, \emptyset, k?v_k), (\{c_3\}, \emptyset, !v_k) \,\} \\
&\quad \text{where condition } c_1 = (\bot, nil),\ c_2 = (e, nil),\ c_3 = (f, nil), \\
&\quad k \text{ is fresh.} \\
[E(v)] &= [[v/x]f] \\
&\quad \text{where } E \text{ is an expression definition and } E(x) \ \triangleq \ f
\end{aligned}$$

4.2 Heaps for the Combinators

$$[f \mid g] = [f] \uplus [g] \tag{17}$$

$$[f \ >x>\ g] = \textsc{pipe}_x([f], [g], \emptyset) \tag{18}$$

$$\begin{aligned}
[g \ \mathbf{where}\ x :\in f] &= \textsc{pipe}_x(\textsc{stop}(F), G(x), G(\overline{x})) \cup G(\overline{x}) \\
&\quad \text{where } F = [f]^{right} \text{ and } G = [g]^{left}
\end{aligned} \tag{19}$$

Figure 2 gives the intermediary and the final heap for the Orc expression $\{let(1) \gg S(x)\}$ **where** $x :\in \{M \mid N\}$. Note the two publications f_1 and f_2, by

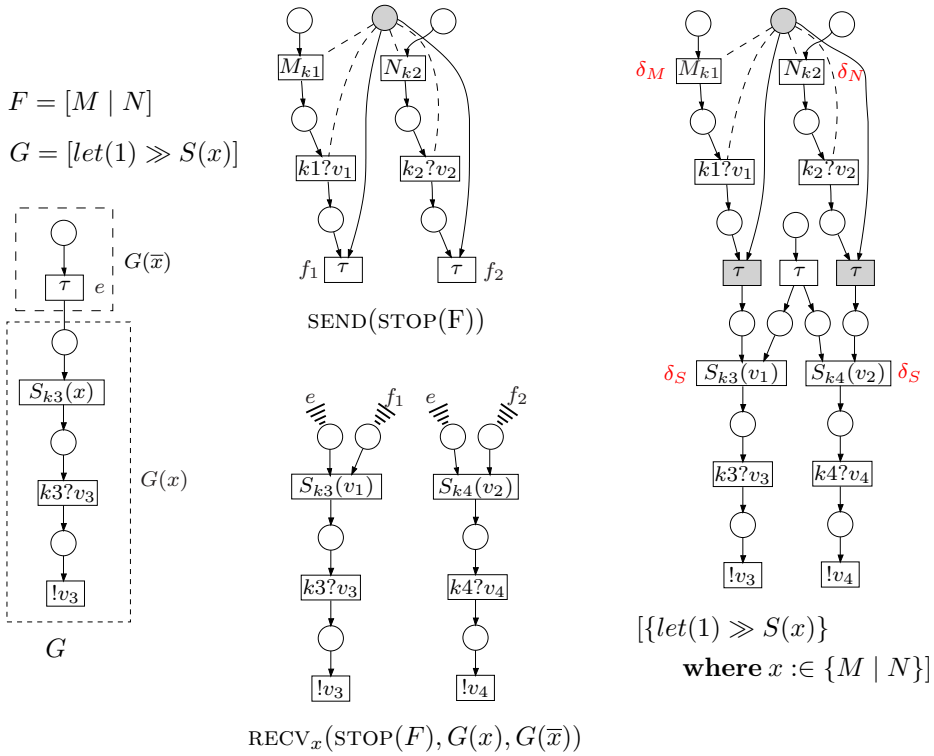

Fig. 2. Heap Construction Example: The shaded condition is the $(\bot, stop)$ condition introduced by the STOP operator. A dashed arrowhead to a minimal condition of the RECV heap from an event name states that the condition depends on that external event. The external events here are e and f_1, f_2 in heaps $G(\overline{x})$ and SEND(STOP(F)) respectively. When these heaps are combined in the right most heap, these events become internal events.

the parallel composition $M \mid N$. These are made conflicting by the extra (shaded) condition created by the STOP operator.

Following Section 2.3, we can now translate the heaps associated to Orc expressions into LAES. The LAES of an expression f is $[\![f]\!] = \mathcal{A}[\,[f]\,]$.

4.3 Recursive Definitions

The treatment of recursive definitions follows that given in [7], except that the denotation of an expression f is the heap $[f]$ instead of the set of traces $\langle f \rangle$. The heap for a recursive Orc definition $f \triangleq Exp(f)$ is the limit of a series of increasing approximations $0 \sqsubseteq Exp(0) \sqsubseteq Exp(Exp(0)) \sqsubseteq \ldots$. To ensure existence of the limit, the least fixpoint of Exp, we show that the Orc combinators are

monotonic with respect to $\sqsubseteq$. For F and G two heaps, define

$$F \prec G \text{ if } F \subseteq G \text{ and } C_F \cap C_{G-F} = \emptyset \tag{20}$$

Then for Orc expressions, $f \sqsubseteq g$ if $[f] \prec [g]$. The motivation for having the second condition in (20) is that it is needed in the proof of Lemma 2 below.

Lemma 1. *Relation $\prec$ is a partial order on heaps.*

Proof. Assume $F \prec G \prec H$. So $C_F \cap C_{G-F} = \emptyset$ and $C_G \cap C_{H-G} = \emptyset$. Writing $H - F = (H - G) \cup (G - F)$, we have $C_F \cap C_{H-F} = (C_F \cap C_{H-G}) \cup (C_F \cap C_{G-F})$. The second term is an empty set. Since $F \subseteq G$, we have $C_F \subset C_G$. This gives $C_F \cap C_{H-G} = \emptyset$ which ensures $F \prec H$ and proves the lemma.

Lemma 2. *The Orc combinators are monotonic in both arguments. In particular, given $f \sqsubseteq g$, then*

$$\begin{gathered} f \mid h \sqsubseteq g \mid h \\ f >x> h \sqsubseteq g >x> h \\ h >x> f \sqsubseteq h >x> g \\ f \textbf{ where } x :\in h \sqsubseteq g \textbf{ where } x :\in h \\ h \textbf{ where } x :\in f \sqsubseteq h \textbf{ where } x :\in g \end{gathered}$$

Proof sketch: (Complete proof in Appendix C). These conditions are established by examining the corresponding constructions on heaps. Monotonicity of most operators can be established by inspection, since they are defined as pointwise functions on the individual events in a heap. One special case is the COPY. $\text{COPY}_l(E, F)$ is not monotonic in its second argument: although $\emptyset \prec F$, it is easy to see that $\text{COPY}_l(E, \emptyset) \not\prec \text{COPY}_l(E, F)$ in general. However, from Section 4.2 we see that we only need monotonicity of the special case where the arguments to COPY are the partition $G(x), G(\overline{x})$ of G. Assume $G \prec G'$ and set $H = G' - G$. We have

$$\begin{aligned} \text{COPY}_l(G'(x), G'(\overline{x})) &= \text{COPY}_l(G(x) \cup H(x), G'(\overline{x})) \\ &= \text{COPY}_l(G(x), G'(\overline{x})) \cup \text{COPY}_l(H(x), G'(\overline{x})) \\ &= \text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x})) \cup \text{COPY}_l(H(x), G'(\overline{x})) \end{aligned} \tag{21}$$

By definition of the copy, $\text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x}))$ is obtained by changing all minimal conditions $c = (e, \mu) \in \text{minConds}(G(x))$ as specified in (14). By the second condition of (20), we have $\text{minConds}(G) \cap C_H = \emptyset$. Thus $\text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x})) = \text{COPY}_l(G(x), G(\overline{x}))$, and thus (23) implies that $\text{COPY}_l(G(x), G(\overline{x})) \prec \text{COPY}_l(G'(x), G'(\overline{x}))$.

5 Correctness of Orc heap semantics

In this section we prove the correctness of the heap semantics for Orc. We do this by showing that the heap semantics is equivalent to an interleaving *trace*

semantics for Orc, developed in [7]. The trace $\langle f \rangle$ of an Orc expression f is a set of the sequence of actions that it can perform. Such a sequence is derived from the labels of successive transitions (according to Figure 1) that f can perform. An additional event, called *substitution event* is introduced to define traces of expressions with free variables. The corresponding rule is

$$f \xrightarrow{[v/x]} [v/x].f$$

which replaces occurrence of variable x in f by the value v.

For the inductive proof, we need to define how configurations of a heap $[f]$ representing an Orc expression f are mapped to traces of $\langle f \rangle$. Since our heap semantics does not introduce substitution events, we need to capture them indirectly. This is done in several steps.

1. We first prepare every heap E as follows: let X be a finite set of free variables containing the set X_E of all free variables of E. For every $x \in X$, let e_x be an additional event not belonging to E, defined as follows:

   $$e_x = (\{c\}, \emptyset, \sigma_x), \quad \text{where } c = (\bot, x)$$

   Then, for every event $e = ({}^\bullet e, \underline{e}, a) \in E$, define the event $e' = ({}^\bullet e', \underline{e}', a)$ where:

   $${}^\bullet e' = {}^\bullet e \cup \{(e_x, e) \mid x \in X \text{ and } e \in E(x)\}, \quad \text{and } \underline{e}' = \underline{e}$$

   Let $E_X = \Phi_1^X(E) = \{e' \mid e \in E\} \cup \{e_x \mid x \in X\}$ be the resulting heap. Each event e_x is concurrent to $E_X - E_X(x)$ and precedes $E_X(x)$. All e_x events are concurrent with each other. Each configuration κ of E gives rise to a set $K_X(\kappa)$ of configurations of E_X through the previously defined map $e \mapsto e'$. Every $\kappa_X \in K_X(\kappa)$ is obtained by adding, to the image κ' of κ, every e_x such that $\kappa(x) \neq \emptyset$, plus possibly additional ones, depending on X.
2. For this step, set X is fixed. Map each configuration κ_X of E_X to the set of all its linear extensions:

   $$\kappa_X \mapsto \ell(\kappa_X) = \left\{\widehat{t} \mid \widehat{t} \text{ is a linear extension of } \kappa_X\right\}$$

   Then, to every $\widehat{t} \in \ell(\kappa_X)$, we associate the following set of traces:

   $$\widehat{t} \mapsto \mathcal{E}(\widehat{t})$$

   where $\mathcal{E}(\widehat{t})$ is the set of all traces obtained as follows: for every value v and every e_x belonging to $\widehat{t}$:
   (a) replace, in e_x, action label σ_x by substitution $[v_x/x]$;
   (b) substitute v_x for x in all actions of $\widehat{t}$ where x occurs;
   (c) replace each event by its associated action.
   By abuse of notation, for $\widehat{T}$ a set of traces, we also write $\mathcal{E}(\widehat{T}) = \bigcup_{\widehat{t} \in \widehat{T}} \mathcal{E}(\widehat{t})$, so that $\mathcal{E} \circ \ell(\kappa_X)$ is well defined. Denote the operations of this step by the map

   $$\Phi_2(\kappa) = \mathcal{E} \circ \ell(\kappa)$$

3. We finally set

$$\Phi_3^X(E) = \bigcup_{\kappa \in \mathbf{Configs}(\Phi_1^X(E))} \Phi_2(\kappa) \quad \text{and} \quad \Phi(E) = \bigcup_{X \supseteq X_E} \Phi_3^X(E)$$

Theorem 2 (Semantic equivalence). *For every Orc expression f, we have*

$$\langle f \rangle = \Phi([f])$$

Proof. The proof is by structural induction over Orc expression f. See appendix D.3.

6 Related Work

Closest to our present study is the work and [14], where Orc expressions are translated to colored Petri net systems [2]. Another closely related work is reported in Bruni et al. [3], where the authors link the Orc language to Petri nets and the Join Calculus; it is advocated that Join Calculus, by offering means to support dynamic creation of names and activities as well as pruning associated with asymmetric conflict, is an adequate formalism for orchestrations. For an approach that focuses on temporal properties without partial orders nor performance evaluation, see [4], where a Timed Automaton semantics of Orc is given and used for verification purposes using the Uppaal tool. On the other hand, a number of papers have been devoted to the Petri net semantics of the most widely used language for orchestration, namely BPEL, see [6,9,13,15,8] and the tutorial [16].

Our work is unique in that it provides a direct coding of a wide area computing language into asymmetric event structures. This is of immediate use in QoS studies, as the latter builds on timed and/or probabilistic enhancements of partial order models [14].

7 Conclusion

We have presented a partial order semantics for Orc, a structured orchestration language with support for termination and recursive process instantiation. The semantics uses *heaps* to encode sets of interrelated events because they simplify manipulation of the fragments of program behavior that arise when analyzing the sub-expressions of a program. These fragments are composed to create effective heaps, from which more traditional asymmetric event structures are derived. We show that the event structure semantics is equivalent to a previous denotational trace semantics.

The heap semantics provides a model of true concurrency and also directly support analysis of non-functional properties of Orc programs, including critical path and dependency analysis that can affect Quality of Service.

A verbatim coding of the Orc heap semantics has been written in Prolog—it takes only two pages of Prolog code. Based on this tool, an analysis of Quality of Service is being developed. Results related to this more applied work will be presented elsewhere.

References

1. Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual Petri nets, Asymmetric Event Structures, and Processes. *Inf. Comput.*, 171(1):1–49, 2001.
2. Eike Best, Raymond R. Devillers, and Maciej Koutny. The Box Algebra = Petri Nets + Process Expressions. *Inf. Comput.*, 178(1):44–100, 2002.
3. Roberto Bruni, Hernán C. Melgratti, and Emilio Tuosto. Translating Orc Features into Petri Nets and the Join Calculus. In *WS-FM*, pages 123–137, 2006.
4. Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of Computation Orchestration via Timed Automata. In *ICFEM*, pages 226–245, 2006.
5. Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.
6. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to petri nets. In *Business Process Management*, pages 220–235, 2005.
7. David Kitchin, William R. Cook, and Jayadev Misra. A Language for Task Orchestration and its Semantic Properties. In *Proc. of the Intl. Conf. on Concurrency Theory (CONCUR)*, 2006.
8. Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.
9. Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing Interacting BPEL Processes. In *Business Process Management*, pages 17–32, 2006.
10. Jean Mairesse and T. Bousch. Asymptotic height optimization for topical IFS, Tetris heaps, and the finiteness conjecture. *J. Amer. Math. Soc*, 15:77–111, 2002.
11. Jean Mairesse and Stéphane Gaubert. Modeling and Analysis of Timed Petri nets using Heaps of Pieces. *IEEE Trans. Autom. Control*, 44(4):683–697, 1999.
12. Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May, 2006. Available for download at `http://dx.doi.org/10.1007/s10270-006-0012-1`.
13. Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *ICSOC*, pages 484–489, 2005.
14. Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Foundations for Web Services Orchestrations: functional and QoS aspects. In *Proceedings International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2006.
15. Franck van Breugel and Mariya Koshkina. Dead-Path-Elimination in BPEL4WS. In *Proceedings of the 5th International Conference on Application of Concurrency to System Design (ACSD)*, pages 192–201, 2005.
16. Franck van Breugel and Mariya Koshkina. Models and Verification of BPEL. Available at http://www.cse.yorku.ca/ franck/research/drafts/tutorial.pdf, 2006. York University.
17. Glynn Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986.

A Proof of Theorem 1

We show that $(\mathcal{G}[E], \preceq, \nearrow, \alpha)$ satisfies all the conditions for an LAES given in section 2.1. Let us first show that $\preceq$ is a partial order on $\mathcal{G}[E]$. By (9) and Condition 4 of Definition 2, $\preceq$ is a preorder on $\mathcal{G}[E]$. It thus remains to show that there exists no non trivial circuit $e_1 \preceq e_2 \preceq \ldots \preceq e_n \preceq e_1$. Let κ be a configuration containing e_1. By Condition 3 of Definition 2, the circuit $e_1 \preceq e_2 \preceq \ldots \preceq e_n \preceq e_1$ must be contained in κ. But, since $\prec \subseteq \nearrow$, we have $e_1 \nearrow e_2 \nearrow \ldots \nearrow e_n \nearrow e_1$, which contradicts Condition 1 of Definition 2. This shows that $\preceq$ is a partial order on $\mathcal{G}[E]$. Also $\lfloor e \rfloor =_{\text{def}} \{e' \mid e' \preceq e\}$ is finite, since an infinite $\prec$ sequence of events would be an infinite $\nearrow$ sequence of events, again contradicting Condition 1 of Definition 2. The same reasoning shows that $\nearrow_{\lfloor e \rfloor} =_{\text{def}} \{(e_1, e_2) \mid e_1, e_2 \in \lfloor e \rfloor,\ e_1 \nearrow e_2\}$ is acyclic. This proves the first statement of the theorem.

To prove the second statement, let $F \subseteq E$ be such that $(F, \preceq, \nearrow, \alpha)$ is an LAES. Denote by κ_F a generic configuration of this LAES. By the definition of configurations, for LAES, any such κ_F must satisfy Conditions 1–3 of Definition 2. In addition, by (9)–(10), κ_F must be such that, for each event e belonging to κ_F, if $f^\bullet \cap {}^\bullet\underline{e} \neq \emptyset$ then $f \in F$. Since $F \subseteq E$, this implies that κ_F also satisfies Condition 4 of Definition 2. Hence κ_F satisfies all conditions of Definition 2 for heap configurations. This proves the theorem.

B Characteristic property of the Stop operator

The following result shows that STOP is a preemption operator.

Lemma 1 *Let E be a heap such that $(\bot, \text{“}stop\text{”}) \notin C_E$ and let $F \subseteq E$. Let bijection φ^{-1} be the inverse map of φ introduced in (13) for the definition of the* $\text{STOP}_F(E)$, i.e *for all $e \in E$,*

$$\varphi^{-1}(e) = \begin{cases} ({}^\bullet e - \{(\bot, \text{“}stop\text{”})\}, \underline{e}, \alpha(e)) & \text{if } e \in F \\ ({}^\bullet e, \underline{e} - \{(\bot, \text{“}stop\text{”})\}, \alpha(e)) & \text{if } e \notin F \end{cases} \tag{22}$$

If κ is a configuration of $\text{STOP}_F(E)$*, then the following properties hold:*

1. *$\varphi^{-1}(\kappa)$ is a configuration of E.*
2. *$\varphi^{-1}(\kappa) \cap F$ contains at most one event; if $\varphi^{-1}(e)$ is such an event, then $\forall f \in \kappa \Rightarrow \neg[e \prec f]$.*

Proof. The first statement is immediate, since φ^{-1} removes read and consume conditions from the preset of each event.

To prove the second statement, assume that $\varphi^{-1}(\kappa) \cap F$ contains two events e and e'. Since $e, e' \in F$, the events $\varphi(e)$ and $\varphi(e')$ have condition $(\bot, \text{“}stop\text{”})$ in their consume preconditions set. From (10), we have that $\varphi(e) \nearrow \varphi(e')$ and $\varphi(e') \nearrow \varphi(e)$, which imply that they cannot both occur in the same configuration κ. Now let $e \in \varphi^{-1}(\kappa) \cap F$ for a configuration κ. Following our previous argument,

$\varphi(e)$ has the condition $(\bot, \text{“}stop\text{”})$ in its consume preconditions set. By definition of STOP, all events f in $\text{STOP}_F(E)$, and hence in κ have $(\bot, \text{“}stop\text{”})$ in their preconditions set ${}^\bullet\underline{f}$. From (10) it follows that $f \nearrow e$ which implies $\neg[e \prec f]$ since e, f are in the same configuration κ.

C Proof of Lemma 2

These conditions are established by examining the corresponding constructions on heaps. The parallel expression $f \mid h$ is monotonic because $\uplus$ is monotonic. $\uplus$ is monotonic because marking E^l is monotonic and $\cup$ is monotonic. Marking is monotonic because it is a pointwise function over minConds, a monotonic selection of a subset of its argument events.

Sequential composition $f >x> h$ is monotonic because $\text{PIPE}_x(E, F, \emptyset)$ is monotonic in both E and F. $\text{PIPE}_x(E, F, \emptyset)$ is monotonic if $\text{SEND}(F)$ and $\text{RECV}_x(E, F, \emptyset)$ are monotonic. SEND, like marking, is a pointwise function over a monotonic selection $!E$ of events from E. Receive $\text{RECV}_x(E, F, \emptyset)$ is trivially monotonic in E, because it is a union over the monotonic subset $!E$, and is monotonic in F if $\text{LINK}(e, v, x, F, \emptyset)$ is monotonic in F. Linking depends on monotonicity of $\text{COPY}_l(F, \emptyset)$, a simple pointwise function on events. Linking also applies a pointwise function based on min, a monotonic subset of a heap.

Monotonicity of asymmetric composition f **where** $x :\in g$ is more complicated. It depends on monotonicity of $G(\overline{x})$ and $\text{PIPE}_x(\text{STOP}(F), G(x), G(\overline{x}))$. The free variable constructs, $G(x)$ and $G(\overline{x})$ are pointwise selectors of events, so they are monotonic. $\text{STOP}(E)$ is also a pointwise function affecting $!E$, a monotonically increasing subset of E. Finally, there is the question of the monotonicity of $\text{PIPE}_x(F, G(x), G(\overline{x}))$. As mentioned above, PIPE_x is monotonic in its first argument, in this case F. Monotonicity of PIPE_x for G depends on monotonicity of $\text{LINK}(e, v, x, G(x), G(\overline{x}))$, which in turn depends on monotonicity of $\text{COPY}_l(G(x), G(\overline{x}))$. Note that $\text{COPY}_l(E, F)$ is not monotonic in its second argument: although $\emptyset \prec F$, it is easy to see that $\text{COPY}_l(E, \emptyset) \not\prec \text{COPY}_l(E, F)$ in general. However, we only need monotonicity of the special case where the arguments to COPY are the partition $G(x), G(\overline{x})$ of G. Assume $G \prec G'$ and set $H = G' - G$. We have

$$\begin{aligned} \text{COPY}_l(G'(x), G'(\overline{x})) &= \text{COPY}_l(G(x) \cup H(x), G'(\overline{x})) \\ &= \text{COPY}_l(G(x), G'(\overline{x})) \cup \text{COPY}_l(H(x), G'(\overline{x})) \\ &= \text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x})) \cup \text{COPY}_l(H(x), G'(\overline{x})) \end{aligned} \tag{23}$$

By definition of the copy, $\text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x}))$ is obtained by changing all minimal conditions $c = (e, \mu) \in \text{minConds}(G(x))$ as specified in (14). By the second condition of (20), we have $\text{minConds}(G) \cap C_H = \emptyset$. Thus $\text{COPY}_l(G(x), G(\overline{x}) \cup H(\overline{x})) = \text{COPY}_l(G(x), G(\overline{x}))$, and thus (23) implies that $\text{COPY}_l(G(x), G(\overline{x})) \prec \text{COPY}_l(G'(x), G'(\overline{x}))$. This finishes the proof of Lemma 2.

D Proof of Correspondence Theorem 2

D.1 Definition of Terms

Linear extension of a trace Consider an execution which is a sequence of events $e_1, \ldots e_n$. The trace t of this execution $\mathcal{E}(e_1, \ldots e_n)$ is the sequence of corresponding actions. Let event e be such that $\forall c \in {}^\bullet\underline{e}$, $\exists\ 1 \leq i \leq n$ such that $c = (e_i, \text{-})$. Then all the events causally preceding e appear in the trace t. Let $1 \leq k \leq n$ be the highest index of these predecessor events. Define the linear extension of such a trace t w.r.t e, t/e as the set of traces

$$t/e = \begin{cases} \mathcal{E}(e_1, \ldots e_k, e, e_{k+1}, \ldots e_n) \\ \mathcal{E}(e_1, \ldots e_k, e_{k+1}, e, \ldots e_n) \\ \qquad\qquad \vdots \\ \mathcal{E}(e_1, \ldots e_k, e_{k+1}, \ldots e_n, e) \end{cases} \tag{24}$$

If we lift the definition of linear extensions of a trace to linear extensions of a set of traces, we have for a configuration $\kappa \in \mathbf{Configs}(\Phi_1^X(E))$ and an event e such that $\kappa \cup \{e\} \in \mathbf{Configs}(\Phi_1^X(E))$,

$$\Phi_2(\kappa \cup \{e\}) = \Phi_2(\kappa)/e$$

D.2 Lemmas

Lemma 2 *For traces t_1, t_2 and a single event action a*

$$(t_1 \mid \{a\}) \mid t_2 \;=\; t_1 \mid (\{a\} \mid t_2) \;=\; (t_1 \mid t_2) \mid \{a\}$$

Proof : Follows directly from the definition of $\mid$ over traces.

Lemma 3 *If trace $s = \mathcal{E}(e_1, \ldots e_n)$ and event e has all its causal predecessors in $\{e_1, \ldots e_n\}$, then for any trace t,*

$$(s \mid t)/e = (s/e) \mid t$$

Proof : Follows directly from the definition of $\mid$ and /over traces.

Lemma 4 *For a heap E and $\kappa \in \mathbf{Configs}(\Phi_1^X(E))$, if $e \in \Phi_1^X(E) - \kappa$ is such that it is concurrent to all events in κ and $\kappa \cup \{e\} \in \mathbf{Configs}(\Phi_1^X(E))$, then*

$$\Phi_2(\kappa \cup \{e\}) = \Phi_2(\kappa) \mid \Phi_2(\{e\})$$

Proof : Since e is concurrent to all events in κ, the linearizations $\Phi_2(\kappa \cup \{e\})$ is all the possible interleavings of linearizations $\Phi_2(\kappa)$ and $\Phi_2(\{e\})$.

Lemma 5 *For two heaps E, F, a fixed set of variables X and distinct marks l, r:*

$$\Phi_1^X(E^l \cup F^r) = \Phi_1^X(E^l) \cup \Phi_1^X(F^r)$$

Proof: Follows from the definition of Φ_1^X. The common events in $\Phi_1^X(E^l)$ and $\Phi_1^X(F^r)$ are the events e_x for all $x \in X$ which also belong to $\Phi_1^X(E^l \cup F^r)$. All the others events are distinct due to the marks l and r and so appear separately in $\Phi_1^X(E^l \cup F^r)$.

As a consequence of this lemma, we have that for any $K \subseteq \Phi_1^X(E^l \cup F^r)$ there exist unique maximal (w.r.t set inclusion) sets $K_1 \subseteq \Phi_1^X(E^l)$ and $K_2 \subseteq \Phi_1^X(F^r)$ such that $K = K_1 \cup K_2$. Moreover if K is a configuration of $\Phi_1^X(E^l \cup F^r)$, then K_1 and K_2 are configurations of $\Phi_1^X(E^l)$ and $\Phi_1^X(F^r)$ respectively, since they satisfy the conditions of Definition 2.

Lemma 6 *For* $\kappa_1 \in \mathbf{Configs}(\Phi_1^X(E^l))$, $\kappa_2 \in \mathbf{Configs}(\Phi_1^X(F^r))$

$$\Phi_2(\kappa_1 \cup \kappa_2) = \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2)$$

Proof: We prove this by induction on the size of the configuration. The base case is obvious when both configurations have a single event. If this event is common to both κ_1 and κ_2 (an event of the form e_x), both sides of the above equation is simply the trace consisting of this single event. If the events are distinct, both sides of the equation are the two traces in which the order of the events are interchanged.

Now suppose that the Lemma holds for configurations $\kappa_1 \in \mathbf{Configs}(\Phi_1^X(E^l))$ and $\kappa_2 \in \mathbf{Configs}(\Phi_1^X(F^r))$ *i.e*,

$$\Phi_2(\kappa_1 \cup \kappa_2) = \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2)$$

Consider the configuration $\kappa_1 \cup \{e\} \in \mathbf{Configs}(\Phi_1^X(E^l))$.

Case 1: e is minimal in κ_1 *i.e* $\nexists\, e' \in \kappa_1$ such that $e \prec e'$. This also means that e is concurrent to all events in κ_1 and is a configuration in itself. From Lemma 4,

$$\Phi_2(\kappa_1 \cup \{e\}) = \Phi_2(\kappa_1) \mid \Phi_2(\{e\})$$

There are two possibilities here. If $e \in \kappa_2$, then e is a substitution event of the kind e_x. Here $\Phi_2(\kappa_1 \cup \{e\} \cup \kappa_2) = \Phi_2(\kappa_1 \cup \kappa_2) = \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2)$ by the hypothesis. Also

$$\begin{aligned}
\Phi_2(\kappa_1 \cup \{e\}) \mid \Phi_2(\kappa_2) &= (\Phi_2(\kappa_1) \mid \Phi_2(\{e\})) \mid \Phi_2(\kappa_2) \quad \text{(Lemma 4)} \\
&= \Phi_2(\kappa_1) \mid (\Phi_2(\{e\}) \mid \Phi_2(\kappa_2)) \quad \text{(Lemma 2)} \\
&= \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2) \quad \text{(since } e \text{ appears in } \kappa_2\text{)} \\
&= \Phi_2(\kappa_1 \cup \{e\} \cup \kappa_2)
\end{aligned}$$

If $e \notin \kappa_2$, then e is concurrent to both κ_1 and κ_2. Hence

$$\begin{aligned}
\Phi_2(\kappa_1 \cup \{e\} \cup \kappa_2) &= \Phi_2(\kappa_1 \cup \kappa_2) \mid \Phi_2(\{e\}) \quad \text{(Lemma 4)} \\
&= (\Phi_2(\kappa_1) \mid \Phi_2(\kappa_2)) \mid \Phi_2(\{e\}) \\
&= (\Phi_2(\kappa_1) \mid \Phi_2(\{e\})) \mid \Phi_2(\kappa_2) \quad \text{(Lemma 2)} \\
&= \Phi_2(\kappa_1 \cup \{e\}) \mid \Phi_2(\kappa_2) \quad \text{(Lemma 4)}
\end{aligned}$$

Case 2: e is not minimal in κ_1. Since all the causal predecessors of e are in κ_1, $\Phi_2(\kappa_1 \cup \{e\}) = \Phi_2(\kappa_1)/e$. Also,

$$\begin{aligned}\Phi_2(\kappa_1 \cup \{e\} \cup \kappa_2) &= \Phi_2(\kappa_1 \cup \kappa_2)/e \\ &= (\Phi_2(\kappa_1) \mid \Phi_2(\kappa_2))/e \quad \text{(Hypothesis)} \\ &= (\Phi_2(\kappa_1)/e \mid \Phi_2(\kappa_2)) \quad \text{(Lemma 3)} \\ &= \Phi_2(\kappa_1 \cup \{e\}) \mid \Phi_2(\kappa_2)\end{aligned}$$

Lemma 7 *For heaps E, F and a set of variables $X \supseteq X_E \cup X_F$*

$$\Phi_3^X(E^l \cup F^r) = \Phi_3^X(E^l) \mid \Phi_3^X(F^r)$$

Proof:

$$\begin{aligned}\Phi_3^X(E^l \cup F^r) &= \bigcup_{\kappa \in \mathbf{Configs}(\Phi_1^X(E^l \cup F^r))} \Phi_2(\kappa) \\ &= \bigcup_{\substack{\kappa_1 \in \mathbf{Configs}(\Phi_1^X(E^l)) \\ \kappa_2 \in \mathbf{Configs}(\Phi_1^X(F^r))}} \Phi_2(\kappa_1 \cup \kappa_2) \quad \text{(Lemma 5)} \\ &= \bigcup_{\substack{\kappa_1 \in \mathbf{Configs}(\Phi_1^X(E^l)) \\ \kappa_2 \in \mathbf{Configs}(\Phi_1^X(F^r))}} (\Phi_2(\kappa_1) \mid \Phi_2(\kappa_2)) \quad \text{(Lemma 6)} \\ &= \left(\bigcup_{\kappa_1 \in \mathbf{Configs}(\Phi_1^X(E^l))} \Phi_2(\kappa_1) \right) \mid \left(\bigcup_{\kappa_2 \in \mathbf{Configs}(\Phi_1^X(F^r))} \Phi_2(\kappa_2) \right) \\ &= \Phi_3^X(E^l) \mid \Phi_3^X(F^r)\end{aligned}$$

Lemma 8

$$\bigcup_{X \supseteq X_{E \uplus F}} (\Phi_3^X(E) \mid \Phi_3^X(F)) = \bigcup_{X \supseteq X_E} \Phi_3^X(E) \mid \bigcup_{X \supseteq X_F} \Phi_3^X(F)$$

Lemma 9 *If trace $t_1[v/x]t_2 \in \Phi(E)$ for a heap E, then $t_1 t_2 \in \Phi([v/x].E)$. Where heap $[v/x].E$ is obtained by replacing variable x by v in the action of all events in E.*

D.3 Correspondence Theorem

For all Orc expressions f,

$$\langle f \rangle = \Phi([f])$$

Proof: The proof is by induction on the structure of f. The set of traces $\langle f \rangle$ differ from those defined in [7] in that we only consider traces where for any variable x, there is *at most* one substitution event $[v/x]$. This restriction is justified since once a variable x has been substituted by a value v, future substitutions for x leave the expression unchanged.

• $\boldsymbol{f = let(v)}$

v is a constant: By definition, $\langle let(v)\rangle = \{\,!v\}$. Since $[let(v)]$ has only one event with the label $!v$, $\Phi([let(v)])$ is $\{\,!v\}$.

v is a variable x: If we consider only the maximal traces $\langle let(x)\rangle = \{[v/x],\, !v\}$ (Other traces are just prefixes of the maximal traces). Since the only event e in $[let(x)]$ has x as a free variable, $\Phi_1^X([let(x)])$ by construction has an event e_x causally preceding e. $\{e_x, e\}$ is the only maximal configuration of $\Phi_1^X([let(x)])$ and $\Phi_2(\{e_x, e\}) = \{[v/x],\, !v\}$. Thus $\Phi([let(x)]) = \langle let(x)\rangle$.

• $\boldsymbol{f = ?k.}$ Considering only maximal traces, $\langle ?k\rangle = \{k?v,\, !v\}$. From section 4.1, $[?k] = \{e_1, e_2\}$ where $e_1 \prec e_2$ and $\alpha(e_1) = k?v,\ \alpha(e_2) = !v$. Clearly $\{e_1, e_2\}$ is the only maximal configuration of $[?k]$ and thus $\Phi([?k]) = \{k?v,\, !v\}$.

• $\boldsymbol{f = M(v).}$

v is a constant: Similar to the proof for $?k$, $\langle M(v)\rangle = \{M_k(v), k?v_k,\, !v_k\}$ and $[M(v)] = \{e_1, e_2, e_3\}$ where $e_1 \prec e_2 \prec e_3$ and $\alpha(e_1) = M_k(v), \alpha(e_2) = k?v_k, \alpha(e_3) = !v_k$.

v is a variable x': Here $\langle M(x)\rangle = \{[v/x]\,\langle M(v)\rangle\}$ Now $[M(x)] = \{e_1, e_2, e_3\}$ where $e_1 \prec e_2 \prec e_3$ and $\alpha(e_1) = M_k(x), \alpha(e_2) = k?v_k, \alpha(e_3) = !v_k$. Similar to the $let(x)$ case, $\Phi_1^X([M(x)]$ would have the additional event e_x causally preceding e_1. Thus $e_x \prec e_1 \prec e_2 \prec e_3$ and $\{e_x, e_1, e_2, e_3\}$ is the only maximal configuration of $\Phi_1^X([M(x)]$. $\Phi_2(\{e_x, e_1, e_2, e_3\})$ is $\{[v/x], M_k(v), k?v_k,\, !v_k\} = \langle M(x)\rangle$.

• $\boldsymbol{f \mid g.}$ Here we need to prove that $\langle f \mid g\rangle = \Phi([f \mid g])$. We have

$$\langle f \mid g\rangle \;=\; \langle f\rangle \mid \langle g\rangle \;=\; \Phi([f]) \mid \Phi([g])$$

from the recursive hypothesis. We thus need to show that

$$\Phi([f \mid g]) \;=\; \Phi([f]) \mid \Phi([g])$$

Now,

$$\begin{aligned}
\Phi([f \mid g]) &= \bigcup_{X \supseteq X_{[f\mid g]}} \Phi_3^X([f \mid g]) \\
&= \bigcup_{X \supseteq X_{[f\mid g]}} \Phi_3^X([f]^{left} \cup [g]^{right}) \\
&= \bigcup_{X \supseteq X_{[f\mid g]}} (\Phi_3^X([f]^{left}) \mid \Phi_3^X([g]^{right})) \quad \text{(Lemma 7)} \\
&= \bigcup_{X \supseteq X_{[f]\uplus[g]}} (\Phi_3^X([f]) \mid \Phi_3^X([g])) \\
&= \bigcup_{X \supseteq X_{[f]}} \Phi_3^X([f]) \mid \bigcup_{X \supseteq X_{[g]}} \Phi_3^X([g]) \quad \text{(Lemma 8)} \\
&= \Phi([f]) \mid \Phi([g])
\end{aligned}$$

• $\boldsymbol{e \in min[\![f >x> g]\!]}$**.** We need to show that $\langle f >x> g\rangle = \Phi([f >x> g])$. We have

$$\langle f >x> g\rangle = \langle f\rangle >x> \langle g\rangle = \Phi([f]) >x> \Phi([g])$$

so it is enough to show that

$$\Phi([f]) >x> \Phi([g]) = \Phi([f >x> g])$$

We successively prove $\subseteq$ (Part 1) and then $\supseteq$ (Part 2). By Definition,

$$[f >x> g] = \textsc{send}([f]) \cup \bigcup_{e \in\, !\,[f], \alpha(e) = !v} \textsc{link}(\tau(e), v, x, [g], \emptyset) \tag{25}$$

Part 1. $\Phi([f]) >x> \Phi([g]) \subseteq \Phi([f >x> g])$

$$\Phi([f]) >x> \Phi([g]) = \{s >x> \Phi([g]) \mid s \in \Phi([f])\}$$

We prove that for any $s \in \Phi([f])$, $s >x> \Phi([g]) \in \Phi([f >x> g])$. We do this by induction on the number of publication events in s.

s has no publication events : This means that the configuration $\kappa \in \mathbf{Configs}(\Phi_1^X([f]))$ for which $s \in \Phi_2(\kappa)$ holds has no publication event either. So $\textsc{send}(\kappa) = \kappa$ is a configuration of $\Phi_1^X(\textsc{send}([f]))$. Since $\textsc{send}([f]) \subseteq [f >x> g]$ (from (25)), $\kappa \in \mathbf{Configs}(\Phi_1^X([f >x> g]))$. Thus $s \in \Phi([f >x> g])$.

$s = r\,!v\,t$, r has no publications: By Definition,

$$s >x> \Phi([g]) = r(t >x> D.\Phi([g]) \mid D.[v/x]\Phi([g]))$$

where D is the sequence of substitutions in r. Let e be the publication event of $[f]$ in the trace s with $\alpha(e) = !v$ and let $f \stackrel{r}{\Rightarrow} f'' \xrightarrow{!v} f' \stackrel{t}{\Rightarrow}$.

Let $\kappa_r \in \mathbf{Configs}(\Phi_1^X([f]))$ be the configuration behind the trace r. Since $\kappa_r \cup \{e\}$ has only one publication event e, $r.\tau = r \in \Phi_2(\kappa'_r \cup \{\tau(e)\})$ where $\kappa'_r \cup \{\tau(e)\}$ is a configuration of $\Phi_1^X(\textsc{send}([f]))$. Similarly, for κ_t the configuration for the trace t in $[f']$, the set of corresponding events κ'_t is a configuration of $\Phi_1^X(\textsc{send}([f']))$. Since $\kappa_r \cup \{e\} \cup \kappa_t$ is a configuration of $\Phi_1^X([f])$,

$$\kappa'_r \cup \{\tau(e)\} \cup \kappa'_t \in \mathbf{Configs}(\Phi_1^X(\textsc{send}([f]))) \tag{26}$$

Now $f' \stackrel{t}{\Rightarrow}$. Since t has one less publication than s, by the initial hypothesis we have

$$t >x> D.\Phi([g]) = t >x> \Phi([D.g]) \in \Phi(f' >x> D.g)$$

i.e any trace $p \in t >x> D.\Phi([g])$ is such that $p \in \Phi_2(\kappa_p)$ where

$$\kappa_p \in \mathbf{Configs}(\Phi_1^X(\textsc{send}([f']) \cup \bigcup_{e' \in\, ![f'], \alpha(e') = !v} \textsc{link}(\tau(e'), v, x, [D.g], \emptyset)))$$

Since κ'_t is the configuration in $\textsc{send}([f']$ that generates trace t,

$$\kappa_p \in \mathbf{Configs}(\Phi_1^X(\kappa'_t \cup \bigcup_{e' \in\, ![f] \setminus e, \alpha(e') = !v} \textsc{link}(\tau(e'), v, x, [D.g], \emptyset))) \tag{27}$$

Combining (26) and (27) and observing that the sequence of substitution events D in in $D.[g]$ is the same as in κ'_r, we have

$$\kappa'_r \cup \{\tau(e)\} \cup \kappa_p \in \mathbf{Configs}(\Phi_1^X(\textsc{send}([f]) \cup \bigcup_{e' \in\, ![f] \setminus e, \alpha(e') = !v} \textsc{link}(\tau(e'), v, x, [g], \emptyset)))$$

and that trace $r.\tau.p = rp \in \Phi_2(\kappa'_r \cup \{\tau(e)\} \cup \kappa_p)$.

Now, consider the heap $\textsc{link}(\tau(e), v, x, [g], \emptyset)$. Since the context heap is empty, this heap is just a copy of $[g]$, with variable x replaced by v and all minimal events having $\tau(e)$ as their causal predecessor. Hence for a trace $q \in D.[v/x].\Phi([g])$, for the corresponding set of events κ_q in $\textsc{link}(\tau(e), v, x, [g], \emptyset)$ is such that $D.q \in \Phi_2(\kappa_q)$. κ_q is not a configuration since its minimal events have the external condition $\tau(e)$ as a predecessor, but $\kappa'_r \cup \{\tau(e)\} \cup \kappa_q$ is a configuration of $\textsc{send}(\kappa \cup \{e\}) \cup \textsc{link}(\tau(e), v, x, [g], \emptyset)$. Moreover since the same substitution events 'D' in $D.q$ occur in r, $rq \in \Phi_2(\kappa'_r \cup \{\tau(e)\} \cup \kappa_q)$.

Finally, it is easy to see that since $\kappa'_r \cup \{\tau(e)\} \cup \kappa_q$ and $\kappa'_r \cup \{\tau(e)\} \cup \kappa_p$ are both configurations, and the events in κ_p and κ_q are concurrent, $\kappa = \kappa'_r \cup \{\tau(e)\} \cup \kappa_p \cup \kappa_q$ is a configuration of such that $r(p \mid q) \in \Phi_2(\kappa)$ where

$$\kappa \in \mathbf{Configs}(\Phi_1^X(\textsc{send}([f]) \cup \bigcup_{e' \in\, ![f] \setminus e, \alpha(e') = !v} \textsc{link}(\tau(e'), v, x, [g], \emptyset)) \cup \textsc{link}(\tau(e), v, x, [g], \emptyset))$$

$$\kappa \in \mathbf{Configs}(\Phi_1^X(\textsc{send}([f]) \cup \bigcup_{e \in\, ![f] \alpha(e) = !v} \textsc{link}(\tau(e), v, x, [g], \emptyset))$$

Hence $r(p \mid q) \in \Phi([f >x> g])$.

Part 2. $\Phi([f]) >x> \Phi([g]) \supseteq \Phi([f >x> g])$

From (25), we see that all minimal events in the LINK heaps are preceded by a $\tau(e)$ event. Hence any configuration of κ of $\Phi_1^X([f >x> g])$ necessarily has events from $\text{SEND}([f])$. We do the proof by induction on the number of such $\tau(e)$ events of $\text{SEND}([f])$ in κ.

κ has no $\tau(e)$ events in $\text{SEND}([f])$ *:* This means that all events in κ belong to $\Phi_1^X(\text{SEND}([f]))$. Clearly, the set of events corresponding to κ in $\Phi_1^X([f])$ is a configuration, with no publication events. Thus for any trace $s \in \Phi_2(\kappa)$, $s \in \Phi_2(\kappa')$ where $\kappa' \in \Phi_1^X([f])$. Thus $s \in \Phi([f])$. Since s has no publish events, $s >x> \Phi([g]) = s$, thus $s \in \Phi([f]) >x> \Phi([g])$.

κ has a $\tau(e)$ event in $\text{SEND}([f])$*:* Let $\kappa'_s = \kappa \cap \Phi_1^X(\text{SEND}([f]))$ be the events from $\text{SEND}([f])$). Let $p \in \Phi_2(\kappa)$ be a trace of $\Phi([f >x> g])$. The restriction of trace p to events in $\Phi_1^X(\text{SEND}([f]))$ is κ'_s. Now, the events corresponding to κ'_s in $[f]$ is a configuration of $\Phi_1^X([f])$. Consider the trace s of this configuration which corresponds to p (s is obtained by replacing all the $\tau(e)$ events in p by e). Clearly s is a trace of $[f]$. Let it be of the form $s = r!vt$, where r has no publication event and D is the sequence of substitutions in r. Let $f \stackrel{r}{\Rightarrow} f'' \stackrel{!v}{\longrightarrow} f' \stackrel{t}{\Rightarrow}$.

Since none of the LINK heap events can occur till a $\tau(e)$ event occurs, trace p starts with the trace r. The next event in s is a publication event e, which corresponds to a $\tau(e)$ event in p. So p can now have events from the heap $\text{LINK}(\tau(e), v, x, D.[g], \emptyset))$. Let κ'_q be the set of events in p from this heap. Any linearization of κ_q q, is a trace in $D.[v/x]\Phi([g])$.

The other possible events in p, κ'_u are the events in $\text{SEND}([f])$ corresponding to the trace t in $[f]$, and events from the corresponding LINK heaps for the $\tau(e)$ events in t. Since $f' \stackrel{t}{\Rightarrow}$, and t has one less $\tau(e)$ event than s, applying the recursive hypothesis we have that for any linearization u of κ'_u, $u \in t >x> D.\Phi([g])$.

Finally, we observe that events in κ'_u and κ'_q are concurrent to each other, and so their possible linearisations are given by $u \mid q$, for all linearisations u, q of κ'_u and κ'_q. Therefore the trace p is of the form $r(u \mid q)$ where $u \in t >x> D.\Phi([g])$ and $q \in D.[v/x]\Phi([g])$.

• ***f* where *x* :∈ *g*.** We need to show that $\langle f \textbf{ where } x :\in g\rangle = \Phi([f \textbf{ where } x :\in g])$, *i.e* to show that

$$\Phi([f]) \textbf{ where } x :\in \Phi([g]) = \Phi([f \textbf{ where } x :\in g])$$

Part 1. $\Phi([f])$ **where** $x :\in \Phi([g]) \subseteq \Phi([f \textbf{ where } x :\in g])$**:** Let $t_1 \in \Phi([f])$ and $t_2 \in \Phi([g])$. We show that $(t_1 \textbf{ where } x :\in t_2)$ when defined, belongs to $\Phi([f \textbf{ where } x :\in g])$. Let $t_1 \in \Phi_2(\kappa_1)$ where $\kappa_1 \in \textbf{Configs}(\Phi_1^{X_1}([f]))$ for some X_1 and $t_2 \in \Phi_2(\kappa_2)$ where $\kappa_2 \in \textbf{Configs}(\Phi_1^{X_2}([g]))$ for some X_2.

Case 1 : t_2 has no publication. This means that configuration κ_2 also has no publication event. Let $\kappa'_2 = \text{SEND}(\text{STOP}(\kappa_2))$. Then, $\kappa'_2 \in \textbf{Configs}(\Phi_1^{X_2}(\text{SEND}(\text{STOP}([g]))))$

and $t_2 \in \Phi_2(\kappa_2')$. Let trace $t_1 = t_1'[v/x]t_1''$ where t_1' has no substitution on x. Clearly the events in t_1' do not depend on x and so $t_1' \in \Phi_2(\kappa_1')$ where $\kappa_1' \in \mathbf{Configs}(\Phi_1^{X_1}([[f](\overline{x})]))$. By definition we have

$$\begin{aligned}
(t_1 \textbf{ where } x :\in t_2) &= t_1' \mid t_2 \\
&\in \Phi_2(\kappa_1') \mid \Phi_2(\kappa_2') \\
&\in \Phi_2(\kappa_1' \cup \kappa_2') \quad \text{(Lemma 6)} \\
&\in \Phi_2(\kappa') \quad \text{where } \kappa' \in \mathbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}([g])) \cup [[f](\overline{x})])) \\
&\quad \text{and } X = X_1 \cup X_2. \text{ (Lemma 5)} \\
&\in \Phi_2(\kappa) \quad \kappa \in \mathbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}(G)) \cup (F(\overline{x})))) \\
&\quad \text{and } F = [f]^{left}, G = [g]^{right} \\
&\in \Phi_2(\kappa) \quad \kappa \in \mathbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}(G)) \cup (F(\overline{x})) \\
&\quad \cup \textsc{recv}_x(\textsc{stop}(G), F(x), F(\overline{x})))) \\
&\in \Phi_2(\kappa) \quad \kappa \in \mathbf{Configs}(\Phi_1^X([f \textbf{ where } x :\in g]) \\
&\in \Phi([f \textbf{ where } x :\in g])
\end{aligned}$$

Case 2 : t_2 has a publication. Let $t_2 = t_2' !v\ t_2''$, $t_2 \in \kappa_2$. Since the subset of κ_2 which generates $t_2' !v$ has only one publish event, $\textsc{stop}(\kappa_2) \in \mathbf{Configs}(\Phi_1^{X_2}(\textsc{stop}\,[g]))$. Clearly $t_2' !v \in \Phi_2(\textsc{stop}(\kappa_2))$ since the event actions remain unchanged. Since the SEND operator only renames publish events to τ, $t_2'\tau \in \Phi_2(\textsc{send}(\textsc{stop}(\kappa_2)))$. Also since the trace discards the τ actions, $t_2' \in \Phi_2(\textsc{send}(\textsc{stop}(\kappa_2)))$ and thus $t_2' \in \mathbf{Configs}(\Phi_1^{X_2}(\textsc{send}(\textsc{stop}(G))))$ where $G = [g]^{right}$. Let e be the τ event which was previously the publication event in κ_2.

Now if $t_1 = t_1'[v/x]t_1'' \in \Phi([f])$, from Lemma 9, we have

$$\begin{aligned}
t_1't_1'' &\in \Phi([v/x].[f]) \\
&\in \Phi([v/x].F) \quad \text{where } F = [f]^{left} \\
&\in \Phi([v/x].(F(\overline{x}) \cup F(x)) \\
&\in \Phi([v/x].(F(\overline{x}) \cup \textsc{copy}_e(F(x), F(\overline{x})))) \\
&\in \Phi(F(\overline{x}) \cup [v/x].\textsc{copy}_e(F(x), F(\overline{x})))
\end{aligned}$$

Now $\textsc{link}(e, v, x, F(x), F(\overline{x}))$ is obtained by adding the event e to the cause of the minimal events in $[v/x].\textsc{copy}_e(F(x), F(\overline{x}))$. Any configuration κ having these events will require event e (and its causal predecessors) to be in κ. Since all the events in $[v/x].\textsc{copy}_e(F(x), F(\overline{x}))$ depend on x, they occur in the subtrace t_1''. Hence the trace $(t_1' \mid t_2')t_1''$ would be a linearization of κ.

$$\begin{aligned}
(t_1 \textbf{ where } x :\in t_2) &= (t_1' \mid t_2')t_1'' \\
&\in \Phi_2(\kappa) \text{ where} \\
&\quad \kappa \in \mathbf{Configs}(\Phi_1^X(F(\overline{x}) \cup \textsc{link}(e, v, x, F(x), F(\overline{x})) \cup \textsc{send}(\textsc{stop}(G))) \\
&\in \Phi_2(\kappa) \text{ where } \kappa \in \mathbf{Configs}(\Phi_1^X([f \textbf{ where } x :\in g]) \\
&\in \Phi([f \textbf{ where } x :\in g])
\end{aligned}$$

Part 2. $\Phi([f])$ **where** $x :\in \Phi([g]) \supseteq \Phi([f \textbf{ where } x :\in g])$**:**

$$[f \textbf{ where } x :\in g] = \textsc{send}(\textsc{stop}(G)) \cup \textsc{recv}_x(\textsc{stop}(G), F(x), F(\overline{x})) \cup F(\overline{x})$$

where $F = [f]^{left}$ and $G = [g]^{right}$. Any two publication events e_1, e_2 in $\textsc{stop}(G)$ - and thus the corresponding τ events $\tau(e_1), \tau(e_2)$ in $\textsc{send}(\textsc{stop}(G))$ - are mutually in conflict, and so they can not appear in the same configuration. Thus any $\kappa \in \textbf{Configs}(\Phi_1^X([f \textbf{ where } x :\in g]))$ will have at most one such event $\tau(e)$ in it. Also, since each of the $\textsc{link}(\tau(e), v, x, F(x), F(\overline{x}))$ heaps in $\textsc{recv}_x(\textsc{stop}(G), F(x), F(\overline{x}))$ has $\tau(e)$ in the causal preset of its minimal events, it follows that κ cannot have events from two such LINK heaps since they would be in conflict.

Case 1 : $\kappa \cap \textsc{send}(\textsc{stop}(G))$ *has no such* $\tau(e)$ *event.* This means that the corresponding configuration in G has no publication event e. Clearly κ does not have events from $\textsc{link}(\tau(e), v, x, F(x), F(\overline{x}))$ since they need $\tau(e)$ to appear in κ. Therefore $\kappa \in \textbf{Configs}(\Phi_1^X(F(\overline{x}) \cup \textsc{send}(\textsc{stop}(G))))$. From Lemma 5, $\kappa = \kappa_1 \cup \kappa_2$ where κ_1 and κ_2 are configurations of $\Phi_1^X(F(\overline{x}))$ and $\Phi_1^X(\textsc{send}(\textsc{stop}(G)))$ respectively. Any trace

$$\begin{aligned}
&t \in \Phi_2(\kappa),\ \kappa \in \textbf{Configs}(\Phi_1^X(F(\overline{x}) \cup \textsc{send}(\textsc{stop}(G))))\\
&\quad \in \Phi_2(\kappa_1 \cup \kappa_2),\ \kappa_1 \in \textbf{Configs}(\Phi_1^X(F(\overline{x})), \kappa_2 \in \textbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}(G)))))\\
&\quad \in \Phi_2(\kappa_1 \cup \kappa_2),\ \kappa_1 \in \textbf{Configs}(\Phi_1^X(F)), \kappa_2 \in \textbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}(G)))))\\
&\quad \in \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2),\ \kappa_1 \in \textbf{Configs}(\Phi_1^X(F)), \kappa_2 \in \textbf{Configs}(\Phi_1^X(\textsc{send}(\textsc{stop}(G)))))\\
&\qquad \text{(Lemma 6)}\\
&\quad \in \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2),\ \kappa_1 \in \textbf{Configs}(\Phi_1^X(F)), \kappa_2 \in \textbf{Configs}(\Phi_1^X(G))\\
&\qquad \text{Since } \kappa_2 \text{ has no publish events, traces } \textsc{send}(\textsc{stop}(\kappa_2)) \text{ remain the same}\\
&\quad \in \Phi_2(\kappa_1) \mid \Phi_2(\kappa_2),\ \kappa_1 \in \textbf{Configs}(\Phi_1^X([f])), \kappa_2 \in \textbf{Configs}(\Phi_1^X([g]))
\end{aligned}$$

Thus $t = t_1 \mid t_2$ where $t_1 \in \Phi([f])$, $t_2 \in \Phi([g])$. Since t_2 has no publication event, $t = t_1$ **where** $x :\in t_2$

Case 2 : $\kappa_1 = \kappa \cap \textsc{send}(\textsc{stop}(G))$ *has a* $\tau(e)$ *event.* e is a publish event in $\textsc{stop}(G)$ and so in G too. Let $\alpha(e) = !v$. Let p be any trace in $\Phi_2(\kappa)$ and t be the trace of the restriction of p to events in κ_1. Since κ_1 is a configuration of $\textsc{send}(\textsc{stop}(G))$, the preemption event $\tau(e)$ is the maximal event in any such linearization t.

From Lemma 1, the events in G corresponding to κ_1, $\kappa_1' = \textsc{stop}^{-1}(\textsc{send}^{-1}(\kappa_1))$ is a configuration of G. Consider the sequence of events in κ_1' which corresponds to the trace t in κ_1. Such a sequence will have only one publication event e which is furthermore the maximal event of the sequence. Since the only event whose label changes in the $\textsc{send}^{-1}(\textsc{stop}^{-1}(\kappa_1))$ transformation is the publication event e, and because $\alpha(\tau(e)) = \tau$, the trace of the sequence of κ_1' events is $t\,!v$, where $\alpha(e) = !v$. Thus $t\,!v \in \Phi(G)$. Since G is simply $[g]^{right}$, we have $t\,!v \in \Phi([g])$.

The other two components of the configuration in κ are the events κ_s from $F(\overline{x})$ and $\kappa_{s'}$ from $\textsc{link}(\tau(e), v, x, F(x), F(\overline{x}))$. Let the sequence of events of κ_s

in p be s and that of $\kappa_{s'}$ be s'. Consider the events in $\kappa_{s'}$. Since they are a copy of events in $F(x)$ with respect to the context $F(\overline{x})$, they have exactly the same causal and conflict relations with events in $F(\overline{x})$ that events in $F(x)$ have. Thus the events $\kappa'_{s'}$ in $F(x)$ corresponding to $\kappa_{s'}$ are such that $\kappa_s \cup \kappa'_{s'}$ is a configuration of $F(x) \cup F(\overline{x})$, *i.e*, F. The LINK heap also replaces all occurrences of the variable x by v in its events.

Consider the sequence of events corresponding to the trace p which belong to $\kappa_s \cup \kappa_{s'}$. Let it be $e_1, \ldots e_{k-1}, e_k, \ldots..e_n$ where e_k is the first event of the sequence that belongs to $\kappa_{s'}$. Let the trace of the sequence $e_1, \ldots e_{k-1}$ be r and that of $e_k, \ldots e_n$ be r'. If $e'_1, \ldots e'_{k-1}, e'_k, \ldots..e'_n$ is the corresponding sequence of events in F (note that e'_k and all other events corresponding to $\kappa_{s'}$ will have x as a free variable here). Now since $\Phi_1^X(F)$ would add the substitution event e_x to the causes of all events depending on x, and because e'_k is the first event in the trace that depends on x, $(e'_1, \ldots e'_{k-1} \mid e_x) e'_k \ldots e'_n$ is a linearization of κ_2 where $\kappa_2 \in \mathbf{Configs}(\Phi_1^X(F))$. In particular, $e'_1, \ldots e'_{k-1}, e_x, e_k, \ldots e'_n$ is a linearization whose trace (sequence of actions) obtained by the transformation $\mathcal{E}$ (see section 5, Step 2.) is $r[v/x]r'$. Thus $r[v/x]r' \in \Phi(F) = \Phi([f])$.

Finally, we note that in trace p, all that events in κ_1 occur before e_k (e_k is the first event from $\kappa_{s'}$ which has $\tau(e)$ as its causal predecessor, and $\tau(e)$ is the maximal event from κ_1 in p). κ_1 events are however concurrent to events in κ_s and thus to $e_1, \ldots e_{k-1}$. Hence such a trace p belongs to $(r \mid t)\ r'$, where $t\,!v \in \Phi([g]$ and $r[v/x]r' \in \Phi([f])$. Thus $p \in \Phi([f])$ **where** $x :\in \Phi([g])$.