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. Developing wide-area distributed applications requires jointly analyzing functional and Quality of Service (QoS) aspects, such as timing properties. Labelled transition systems and sequential trace semantics - the common semantic domains - do not facilitate this kind of analysis because they do not precisely express the causal relationships between events. *Asymmetric Event Structures* (AES) provide an explicit representation of the causal dependencies between events in the execution of a system and allow for an elegant coding of preemption. Event structures are, however, difficult to construct compositionally, because they cannot easily represent fragments of a computation. The *heaps* we develop here allow for such a representation, and easily generate AES. In this paper, we develop a partial-order semantics in terms of heaps, for Orc, an orchestration language used to describe distributed computations over the internet. We briefly show how Orc, and this new semantics, are used for QoS studies of wide area orchestrations.

1 Introduction

Orchestrating Web services consists of a combination of different activities.

A primary concern is to ensure that the expected functionality is indeed correctly implemented. This requires semantic studies for the formalisms used in specifying the functional aspect of Web service orchestrations. Examples of such studies include the translation of the industrial standard BPEL into WorkFlow nets [18] (a special subclass of Petri nets) or the pi-calculus [14], from which analysis techniques and tools for BPEL [13,2] were developed.

Another important, yet much less addressed task consists in ensuring that the Web service orchestration offers the due Quality of Service (QoS). QoS parameters are not firmly established, but they typically include response time (latency), availability, maximum allowed query rate (throughput), and security. The Web Service Level Agreement (WSLA) framework [11] is a standard proposed by IBM for QoS parameters in Web Services. When applied to the management of OEM/supplier cooperations, orchestrations must make precise the

[⋆] This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003), DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

duties and responsibilities of the different actors in such chains, via contracts [5]. Having contracts with each subcontractor, the orchestration can establish the overall contract with its customers. This process is called contract composition.

We believe there is a need for semantic studies underpinning the design of Web services orchestrations in all its aspects: functional, QoS, and contracts, including contract composition. Developing such a holistic approach can become quickly cumbersome if rich formalisms for describing Web services orchestrations are considered, such as, e.g., BPEL. The functional semantics of BPEL is in itself complex, due to the large number of features offered. Extending such semantics to encompass QoS aspects can be cumbersome. Orc [12] has been recently proposed as a small and elegant language for wide area computing and Web services orchestrations. While keeping small, it offers the main features required by wide area computing, namely: service call, parallel and sequential composition, preemption, and recursion. Orc has been successfully used to model typical workflow patterns defined by Van der Aalst et al [1,8].

This paper proposes the foundations for an Orc based design of Web services orchestrations, including both functional and QoS aspects, and supporting contract composition. An interleaving semantics, both operational and denotational, was proposed for Orc in [12]. To prepare for a combined functional/QoS use, we propose in this paper a partial order semantics that keeps track of causalities and concurrency. This allows us to address all the aspects of QoS where causality and concurrency relating the different site calls matters. For example, if an orchestration causally depends on a given site call, failure of this site to deliver proper service causes failure of the orchestration. Another example is that of latency: causal dependencies and concurrency between site calls and other events are reflected into the dates of completion of these different events. Companion paper [16] details the use of this semantics for QoS studies and contract composition, and describes the resulting *TOrQuE* tool (**T**ool for **Or**chestration **Qu**ality of Service **E**valuation).

The paper is organized as follows. Section 2 briefly introduces Orc and its operational semantics. Asymmetric event structures and heap semantics of Orc are described in Section 3, where its use in QoS studies is sketched. Related work is given in Section 4.

2 Orc Overview

An Orc program consists of a set of definitions and a *goal* expression which is to be evaluated. Orc assumes that basic services, like sequential computation and data manipulation, are implemented by primitive *sites*. Orc provides constructs to orchestrate the concurrent invocation of sites.

The syntax of Orc is given in the upper portion of figure 1. Orc defines three basic operators. For Orc expressions f, g, "$f \mid g$" executes f and g in parallel. "$f >x> g$" evaluates f first and for *every* value returned by f, a *new* instance of g is launched with variable x assigned to this return value. "f **where** $x :\in g$" executes f and g in parallel. When g returns its *first* value, x is assigned to this value and the computation of g is terminated. All site calls in f having x as a parameter are blocked till x is defined ($i.e$, till g returns its first value).

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

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

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

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

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

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

$$\frac{[\![E(x) \quad \underline{\Delta} \quad f\;]\!] \;\in\; D}{E(p) \;\xrightarrow{\tau} [p/x].f}\quad (\textsc{Def})$$

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

The operational semantics of Orc is given in Figure 1 [12], using SOS rules. 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:

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

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.

The `CarOnLine` toy example. `CarOnLine` is a composite service for buying cars online, together with credit and insurance. A simplified schematic description of the service is given in figure 2. On receiving a car model as an input

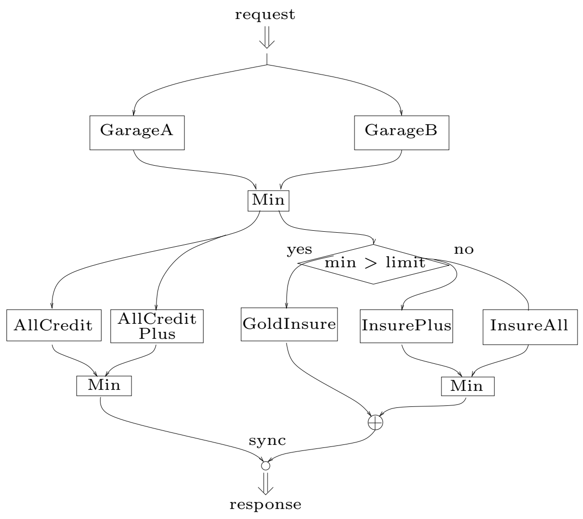

Fig. 2. A simplified view of the CarOnLine orchestration. The calls to GarageA and GarageB are guarded by a timer that returns a "Fault" message at timeout.

query, the CarOnLine service first sends parallel requests to two car dealers (GarageA,GarageB), getting quotations for the car. We guard the calls to each garage by a timer, which kills the waiting when timeout occurs. The best offer (minimum price) is selected and credit and insurances are parallely found for the offer. Two banks (AllCredit,AllCreditPlus) are queried for credit rates and the one offering a lower rate is chosen. For insurance, if the car price of the best offer is greater than a certain limit, any insurance offer by service GoldInsure is accepted. If not, two services (InsurePlus,InsureAll) are parallely called and the one offering the lower insurance rate is chosen. In the end, the (car-price,credit-rate,insurance-rate) tuple is returned to the requestor.

The Orc program for CarOnLine is given in Figure 3. *CarPrice* parallelly calls *GarageA* and *GarageB* for quotations. Calls to these garages are guarded by a timer site *Timer* which returns a fault value T time units after the calls are made. The *let* site simply returns the values of its arguments—sites can only execute when all their parameters are defined and thus can be used to synchronize parallel threads. The value returned by *CarPrice* (here the variable p) is passed as argument to *GetCredit* and *GetInsur* which parallelly find credit and insurance rates for the price.

$$
\begin{array}{lll}
\mathit{CarOnline}(\mathit{car}) & =_{\mathrm{def}} & \mathit{CarPrice}(\mathit{car})\ >p>\ \mathit{let}(p,c,r) \\
 & & \quad \textbf{where}\ \ c :\in \mathit{GetCredit}(p) \\
 & & \qquad\qquad\ \ r :\in \mathit{GetInsur}(p) \\
\mathit{CarPrice}(\mathit{car}) & =_{\mathrm{def}} & \{\mathit{GuardedMin}(p1,p2) \\
 & & \qquad \textbf{where}\ \ p1 :\in \mathit{GarageA}(\mathit{car}) \mid \mathit{Timer}(T) \\
 & & \qquad\qquad\qquad p2 :\in \mathit{GarageB}(\mathit{car}) \mid \mathit{Timer}(T)\} \\
 & & >p>\ \ \{\mathit{if}(p \neq \mathit{Fault})) \gg \mathit{let}(p)\} \\
\mathit{GetCredit}(p) & =_{\mathrm{def}} & \mathit{Min}(r1,r2) \\
 & & \qquad \textbf{where}\ \ r1 :\in \mathit{AllCredit}(p) \\
 & & \qquad\qquad\qquad r2 :\in \mathit{AllCreditPlus}(p) \\
\mathit{GetInsur}(p) & =_{\mathrm{def}} & \{\mathit{if}(p \geq \mathit{limit}) \gg \mathit{GoldInsure}(p)\} \\
 & & \mid \\
 & & \{\mathit{if}(p \leq \mathit{limit}) \gg\ \ \mathit{Min}(ip,ia) \\
 & & \qquad\qquad \textbf{where}\ \ ip :\in \mathit{InsurePlus}(p) \\
 & & \qquad\qquad\qquad\qquad ia :\in \mathit{InsureAll}(p)\}
\end{array}
$$

Fig. 3. CarOnLine in Orc. *GuardedMin* takes the minimum of the values received before timeout and otherwise returns *Fault*.

3 Event structure semantics of Orc

In this section we describe our partial order semantics. We first recall asymmetric event structures, and then introduce heaps.

3.1 Asymmetric Event Structures

Following [19,3], 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 $\lfloor e \rfloor =_{\mathrm{def}} \{e' \in E \mid e' \preceq e\}$ is finite;
2. $\forall e, e' \in E$:

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

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

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

where $\#^a$ is the *conflict relation*, recursively defined by:

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

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

By abuse of notation, we write $e \#^a f$ to mean $\#^a(\{e, f\})$. Condition (5) ensures that a conflict with e is inherited by all the events caused by e. 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. Although event structures are a convenient semantic domain for complete programs, they cannot represent fragments thereof, which arise naturally when constructing the behavior of a program from its sub-parts. By offering the additional concept of *place*, Petri nets and their extensions make composition and structural translation easier. Explicit encoding of places allows one fragment to depend upon resources supplied by another fragment. Petri nets with read arcs also allow us to elegantly code the *preemption* behaviour in Orc's **where** operator: the first "publish" event prevents all subsequent events from occuring. To bypass the nontrivial construction of Petri nets supporting recursion, we chose to generate directly a particular representation of unfoldings of nets with read arcs, which we call *heaps*. Heaps can then be easily translated into event structures and allow for easy coding into software.

3.2 Heaps

Heaps are sets of labeled events coded in a particular form, following an original idea of Esparza et al. [10]. A heap event possesses a label—the Orc action it represents—and is characterized by the conditions that enable its occurrence. These enabling conditions can either be consumed by the event or can be read and not consumed. Each condition, in turn, refers to the event that created it. Marks are used to distinguish different conditions created by the same event.

More precisely, we are given two underlying sets $\mathcal{A}$ of *labels* and $\mathcal{M}$ of *marks*, and a special element $\star \in \mathcal{A}$, to be interpreted as the initialization action. Sets $\mathcal{E}$ of all events and $\mathcal{C}$ of all conditions are inductively defined as follows:

- $\bot = (\emptyset, \emptyset, \star) \in \mathcal{E}$;
- if $f \in \mathcal{E}$ and $\mu \in \mathcal{A}$, then $c = (f, \mu) \in \mathcal{C}$; μ is the *mark* of c;
- for $\mathbf{c}$ and $\mathbf{c}'$ two subsets of $\mathcal{C}$ such that $\mathbf{c} \cap \mathbf{c}' = \emptyset$ and $\mathbf{c} \cup \mathbf{c}' \neq \emptyset$, then $e = (\mathbf{c}, \mathbf{c}', a) \in \mathcal{E}$; a is the *label* of e; ${}^{\bullet}e =_{\text{def}} \mathbf{c}$ and $\underline{e} =_{\text{def}} \mathbf{c}'$ are the set of conditions *consumed* and *read* by e, and ${}^{\bullet}\underline{e} =_{\text{def}} {}^{\bullet}e \cup \underline{e}$ is the *preset* of e.

Definition 1. *A heap is a tuple (E, C, S, A, M), where $A \subseteq \mathcal{A}$, $M \subseteq \mathcal{M}$, $E \subseteq S \subseteq \mathcal{E}$, and $C \subseteq \mathcal{C}$ are such that $\bot \in E$ and*

$$e = (\mathbf{c}, \mathbf{c}', a) \in E \Rightarrow \mathbf{c} \cup \mathbf{c}' \subseteq C \text{ and } a \in A$$
$$c = (f, \mu) \in C \Rightarrow f \in S \text{ and } \mu \in M$$

E is the set of events *of the heap and S is its* support*. For $f \in S$, set $f^\bullet =_{\mathrm{def}} \{c \in C \mid \exists \mu \in M, c = (f, \mu)\}$. Define the set of* minimal conditions *of E to be* $\mathrm{minConds}(E) =_{\mathrm{def}} \{c \in C \mid c = (f, \mu)$ *for* $f \notin E\}$.

We identify the heap with its set of events E. Support S allows for conditions to be caused by events not belonging to E. With this non classical notion of support, heaps can model program fragments (unlike event structures).

Given a heap E we define the following relations between events in E (superscript * denotes transitive closure):

$$\preceq_E = \lhd^* \text{ where } \lhd = \{(f,e) \mid f^\bullet \cap {}^\bullet\underline{e} \neq \emptyset\} \ \cup \ \{(e,e) \mid e \in E\} \tag{6}$$

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

$$\nearrow_E = \nearrow'_E \ \cup \{(e,f) \mid e \#^a_E f\} \tag{7}$$

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 the pair $(\preceq_E, \nearrow'_E)$ satisfies conditions (1) and (2), 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

$$\alpha_E(e) =_{\mathrm{def}} a \tag{8}$$

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

$$\min(E) = \{e \in E \mid \forall f \in E : f \preceq_E e \Rightarrow f = e\} \tag{9}$$

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 4, $e \preceq f_1$ 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 $f \in E$.*

Heap configurations represent self-enabled executions. By condition 3, condition 4 is equivalent to $f \in \kappa$. Conditions 1–3 coincide with those involved in the definition of configurations for AES. Condition 4 is new; it amounts to requiring that κ needs no external event from the support, for its enabling. Let **Configs**(E) be the set of all configurations of heap E.

One may expect $(E, \preceq, \nearrow, \alpha)$ to be an LAES. This is not true in general. The reason is that heaps can represent program fragments, whereas LAES don't. 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:*

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

Say that heap E is effective *if $\mathcal{G}[E] = E$ holds.*

$\mathcal{G}[E]$ 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 (6), (7), and (8), respectively.

Theorem 1 ([17]). *$\mathcal{A}[E] = (\mathcal{G}[E], \preceq, \nearrow, \alpha)$ is an LAES. Furthermore, $\mathcal{G}[E]$ is the maximal subset of events of E that induces an LAES.*

Heaps will be used to give the semantics of fragments of Orc programs, i.e., programs requiring a context. This allows for a structural construction of the semantics of Orc. Effective heaps will represent Orc programs that are self-enabled and can be executed.

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:* for E and F heaps, and *left* and *right* fixed marks:

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

– *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 $\varphi(e)$ as follows:

$$\varphi(e) =_{\text{def}} \begin{cases} ({}^\bullet e \cup \{(\bot, \mathit{stop})\}, \underline{e}, a) & \text{if } e \in F. \\ ({}^\bullet e, \underline{e} \cup \{(\bot, \mathit{stop})\}, a) & \text{if } e \notin F. \end{cases} \tag{10}$$

– *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:

$$(e, \mu) = \begin{cases} (e, (\mu, l)) & \text{if } (e, \mu) \notin C_F \\ (e, \mu) & \text{if } (e, \mu) \in C_F \end{cases} \tag{11}$$

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.

3.3 The heap semantics of Orc

In this section, we construct the heap semantics of Orc in a structural way. Some intermediate steps will require heaps that are not effective. Heaps of well formed Orc expressions will all be effective, however, thus giving rise to an LAES.

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

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

  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:

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

- *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:

$$\alpha(e) = \begin{cases} \tau_x \text{ if } \alpha(e) = !x, \text{ for any variable } x \\ \tau \text{ otherwise} \end{cases} \tag{12}$$

  The heap $\textsc{send}(E)$ 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,

$$\textsc{link}(f, v, x, E, C)$$

  is a (non effective) heap in which variable x is bound to value v after external event f. The *context* heap C identifies parts of E that are not affected by the variable binding. $\textsc{link}(f, v, x, E, C)$ is the heap obtained as follows:
  1. Create $E' = \textsc{copy}_f(E, C)$ 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 $e = \varphi_f^{-1}(e')$:

$$e' = \begin{cases} ({}^\bullet e' \cup \{(f, e)\}, \underline{e}', [v/x]a) & \text{if } e' \in \min(E') \\ ({}^\bullet e', \underline{e}', [v/x]a) & \text{if } e' \notin \min(E') \end{cases} \tag{13}$$

  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. The heap constructed here does not contain the event f referred by $e' \in \min(E')$.

– *Receive:* We next construct a (non effective) heap that can receive any value published by another heap. If e is a publication event, $\tau(e)$ is the event e with its action changed according to (12). We define

$$\text{RECV}_x(E, F, C) = \bigcup_{f \in !E, \alpha(f) = !v} \text{LINK}(\tau(f), v, x, F, C)$$

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

– *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.

$$\text{PIPE}_x(F, G, C) = \text{SEND}(F) \ \cup \ \text{RECV}_x(F, G, C)$$

Heaps of Base Expressions. For Orc expression f, $[f]$ is its heap denotation. In the following, symbol *nil* indicates the absence of mark.

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

Heaps for the Combinators.

$$\begin{aligned}
[f \mid g] &= [f] \uplus [g] \\
[f > x > g] &= \text{PIPE}_x([f], [g], \emptyset) \\
[g \ \mathbf{where}\ x :\in f] &= \text{PIPE}_x(\text{STOP}(F), G(x), G(\overline{x})) \cup G(\overline{x}) \\
&\quad \text{where } F = [f]^{right} \ \text{ and } G = [g]^{left}
\end{aligned}$$

Theorem 2 **([17]).** *Heaps of base expressions are all effective. If $[f]$ and $[g]$ are effective heaps, then so are their compositions via the above three combinators.*

Recursive Definitions. The treatment of recursive definitions follows that given in [12], 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 $f \ \underline{\Delta} \ Exp(f)$ 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

$$F \prec G \ \text{ if } \ F \subseteq G \text{ and } C_F \cap C_{G-F} = \emptyset \tag{14}$$

Then for Orc expressions, $f \sqsubseteq g$ if $[f] \prec [g]$. The motivation for having the second condition in (14) is that it is needed in the proof of Lemma 2 below.

Lemma 1 ([17]). *Relation $\prec$ is a partial order on heaps.*

Lemma 2 ([17]). *The Orc combinators are monotonic in both arguments. In particular, given $f \sqsubseteq g$, then*

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

Complete proofs of the theorems and lemmas is given in [17], along with a correctness proof of this semantics, with respect to the semantics of Figure 1.

3.4 Examples

Figure 4 gives the intermediary and the final heap for the Orc expression

$$\{let(1) \gg S(x)\} \textbf{ where } x :\in \{M \mid N\}.$$

Note the two publications f_1 and f_2, by the parallel composition $M \mid N$. These are made conflicting by the extra (shaded) condition created by the STOP operator. We show in the middle two intermediate steps of the translation. Subexpression $F = M \mid N$ has two emissions, by M and N respectively. By Rule (ASYM1V) of Figure 1, F, when used in the **where** context, must be terminated just after its first publication event f_1 or f_2. This is realized by the SEND(STOP(F)) mechanism; the shaded condition create asymmetric conflict causing the first publication to preempt the other one.

The second heap named RECV$_x(\dots)$ properly puts G in the two conflicting contexts of publication events f_1 or f_2. A dashed arrowhead to a minimal condition of the 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, thus showing that the resulting final heap is effective.

The `CarOnLine` toy example, continued. Figure 5 shows a diagram of the event structure corresponding to the CarOnLine program written in Orc. The event structure is generated by our tool and it collects all the possible executions of CarOnLine, taking into account timers and other interactions between data and control. Each execution has the form of a partial order and can be analysed to derive appropriate QoS parameter composition, for each occurring pattern. Each site call to a service M is translated into three events, the *call* (M), the *call return* ($?M$) and the *publish action* (!), which lengthens the structure.

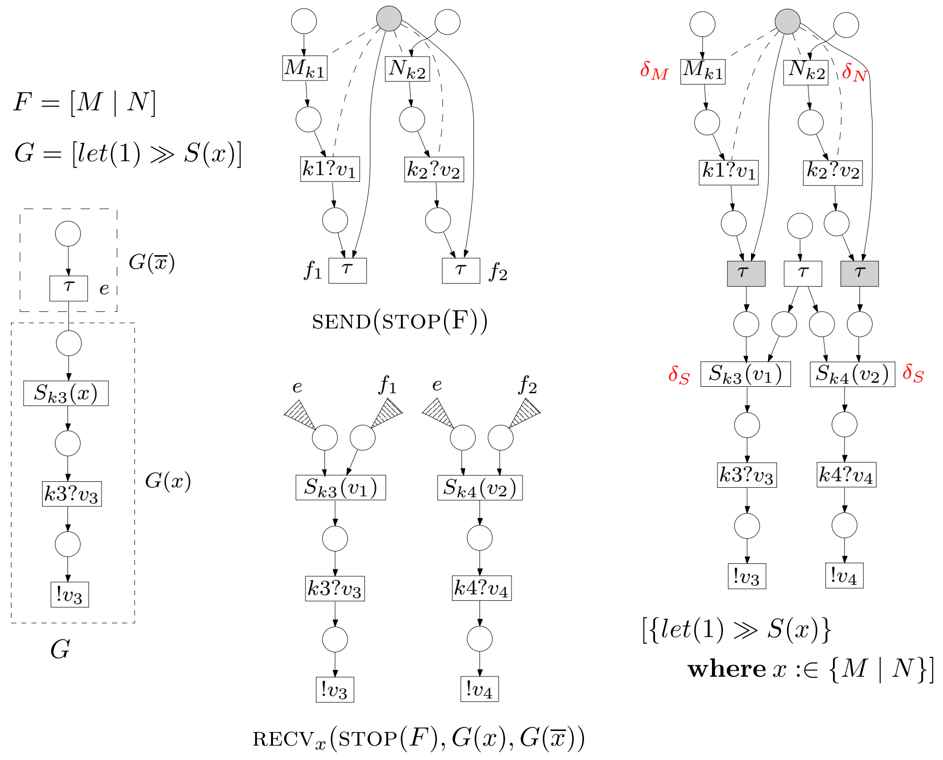

Fig. 4. Heap semantics of the Orc expression $\{let(1) \gg S(x)\}$ **where** $x :\in \{M \mid N\}$. Solid/dashed arcs point back to consumed/read conditions. Dashed arrow heads point back to causes not belonging to the considered heap—this is the way program fragments are captured. The red color refers to QoS aspects, see Section 3.5.

3.5 QoS studies on Orc

Having the event structure semantics of Orc allows us to address all the aspects of QoS where causality relating the different site calls matters. As an example, we focus on latency, depicted in red in Figure 4. We assign to web service calls M, N and S a latency represented by variables δ_M, δ_N and δ_S respectively. Given outcomes for δ_M, δ_N, and δ_S, we get the overall latency δ_E for the orchestration $E = \{let(1) \gg S(x)\}$ **where** $x :\in \{M \mid N\}$, by using its heap in Figure 4. This heap exhibits two maximal configurations, which correspond to M or N publishing first : these two publish events (the shaded τ events) are in conflict. The resolution of this conflict is driven by the actual value for δ_M and δ_N : for *e.g*, if $\delta_M < \delta_N$, S_{k3} will occur (but S_{k4} will not). For each configuration, we add the latencies along each causality path, and consider the maximum latency of all the incoming paths at a synchronization event. Here, when $\delta_M < \delta_N$, the overall latency will thus be $\delta_M + \delta_S$. An important fact is that latency and conflict mutually interact: who publishes first has a consequence on which configuration is actually executed, which in turn has a consequence on the overall latency. Note that this analysis also supports the use of timeouts in the orchestration to guard the waiting for answers to site calls.

4 Related Work

Closest to our present study is the work [15], where Orc expressions are translated to colored Petri net systems [4]. Bruni, Melgratti and Tuosto [7] link the Orc language to Petri nets and the join calculus. Together with the event structure semantics for nominal calculi given in Bruni, Melgratti and Montanari [6], this yields a chain of transformations that yield an event structure semantics for suitable Orc programs. However, [6] focusses on the subclass of persistent grammars, which avoids the use of asymmetric conflicts. We consider asymmetric conflict as central for dealing with orchestration dynamics; in fact, preemption-based constructs such as timeouts, races etc. inevitably lead to asymmetric conflicts not covered by prime event structures, see figure 4. For an approach that focuses on temporal properties without partial orders nor performance evaluation, see [9], where a Timed Automaton semantics of Orc is given and used for verification purposes using the Uppaal tool.

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 [15,16].

5 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.

The heap semantics provides a model of true concurrency and also directly support analysis of non-functional properties of Orc programs. In [16] some of the authors develop a theory of "soft" contracts in which Service Level Specifications (SLS) are expressed in terms of probability distributions on QoS parameters. Monte-Carlo simulations of the orchestration provide a simple approach to compose these probabilistic contracts. Each simulation is an execution of the orchestration's heap in which latencies of the calls to services are drawn from the corresponding contract's probability distribution. Using the technique given in section 3.5 to compose latencies, the empirical probability distribution for the overall orchestration latency is derived.

References

1. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.
2. Jesús Arias-Fisteus, Luis Sánchez Fernández, and Carlos Delgado Kloos. Applying model checking to BPEL4WS business collaborations. In *SAC*, pages 826–830, 2005.
3. Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual Petri nets, Asymmetric Event Structures, and Processes. *Inf. Comput.*, 171(1):1–49, 2001.
4. Eike Best, Raymond R. Devillers, and Maciej Koutny. The Box Algebra = Petri Nets + Process Expressions. *Inf. Comput.*, 178(1):44–100, 2002.
5. Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. SLA management in federated environments. *Computer Networks*, 35(1):5–24, 2001.
6. Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Event structure semantics for nominal calculi. In *CONCUR*, pages 295–309, 2006.
7. 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.
8. William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in orc. In *COORDINATION*, pages 82–96, 2006.
9. Jin Song Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of Computation Orchestration via Timed Automata. In *ICFEM*, pages 226–245, 2006.
10. Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.
11. Alexander Keller and Heiko Ludwig. The wsla framework: Specifying and monitoring service level agreements for web services. *J. Network Syst. Manage.*, 11(1), 2003.
12. David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR*, pages 477–491, 2006.
13. C. Ouyang, E. Verbeek, W.M.P van der Aalst, and S. Breutel. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-15, BPMcenter.org, 2005.
14. Frank Puhlmann and Mathias Weske. Using the *pi*-Calculus for Formalizing Workflow Patterns. In *Business Process Management*, pages 153–168, 2005.
15. Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Foundations for Web Services Orchestrations: functional and QoS aspects. In *Proceedings ISOLA 2006*, 2006.
16. Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Probabilistic qos and soft contracts for transaction based web services. In *ICWS*, pages 126–133, 2007.
17. Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, and Claude Jard. Event Structure Semantics of Orc. IRISA Internal Report No. 1853, June 2007. Available for download at `http://www.irisa.fr/distribcom/benveniste/pub/heaps4Orc2007.pdf`.
18. Wil M. P. van der Aalst and Twan Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. In *ICATPN*, pages 62–81, 1997.
19. Glynn Winskel. Event Structures. In *Advances in Petri Nets*, pages 325–392, 1986.

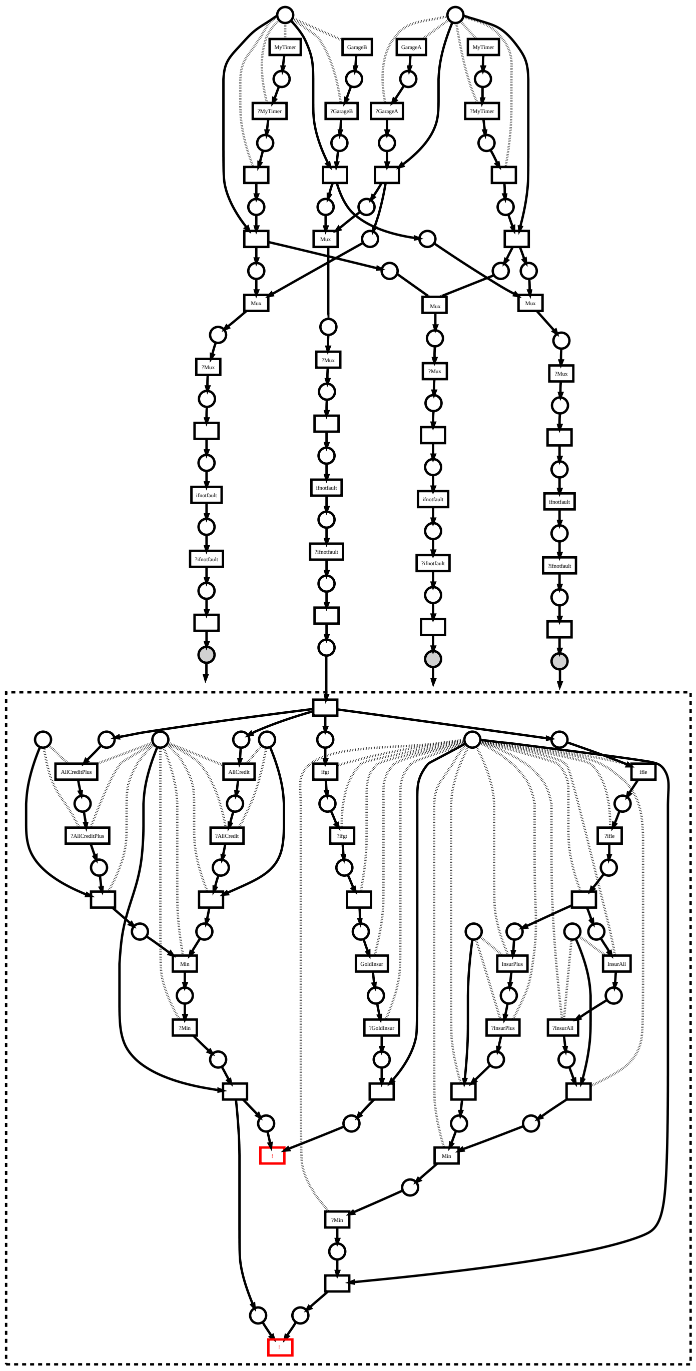

Fig. 5. A labelled event structure collecting all possible executions of CarOnLine, as generated by our tool. The three dangling arcs from the shaded places are followed by copies of the boxed net.