Tracking Causal Dependencies in Web Services Orchestrations Defined in ORC

Matthieu Perrin, Claude Jard, Achour Mostefaoui

Tracking Causal Dependencies in Web Services Orchestrations Defined in ORC

Matthieu Perrin, Claude Jard, and Achour Mostéfaoui

Université de Nantes, LINA, 2 rue de la Houssinière, F-44322 Nantes, France
"first name"."last name"@univ-nantes.fr **

Abstract. This article shows how the operational semantics of a language like ORC can be instrumented so that the execution of a program produces information on the causal dependencies between events. The concurrent semantics we obtain is based on asymmetric labeled event structures. The approach is illustrated using a Web service orchestration instance and the detection of race conditions.

1 Introduction

Several languages have been proposed to program applications based on Web service orchestrations (BPEL [1] is probably one of the best known). The present work is based on Orc [2, 3], an orchestration language whose definition is based on a mathematical semantics, which is needed to define precisely the notion of causality. Orc is designed over the notion of *sites*, a generalization of functions that can encapsulate any kind of externally defined web sites or services as well as Orc expressions. As usual for languages, the operational semantics of Orc was defined as a labeled transition system. Such semantics produces naturally sets of sequential traces, which explicitly represent the observable behaviors of an Orc program [4].

Finding the causal dependencies in a program is very useful for error detection. In a non-deterministic concurrent context, this analysis cannot be based solely on the static structure of the program and requires execution. Dependencies are also very difficult to extract from a sequential record without additional information to unravel the interleaving of events. This is especially true for the analysis of QoS or of non functional properties, like timing constraints derived from the critical path of dependencies [5]. We consider any Orc program, which has been already parsed and expanded into its Orc calculus intermediate form. In this program, we distinguish the actions, which are the site calls, and the

** This work has been partially supported by a French government support granted to the CominLabs excellence laboratory (Project *DeSceNt:Plug-based Decentralized Social Network*) and managed by the French National Agency for Research (ANR) in the "Investing for the Future" program under reference Nb. ANR-10-LABX-07-01. It was also partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003).

publications (return values of expressions). An event is the occurrence of an action during the execution of the Orc program. The events are linked by causal dependencies, that force the events to be executed in a certain order. We can distinguish three kinds of dependencies:

- the dependencies that are imposed by the control flow of the program defined by the semantics of the Orc combinators and imposed by the binding mechanism of Orc variables;
- the dependencies that are provided by the server executing the site calls. These external dependencies are not part of the Orc description, but could be returned by the site. We will consider at least that the possible return of a site call is directly caused by this call;
- the dependencies induced by preemption (the pruning operator of Orc).

The method used in this article is to extend the standard structural operational semantics (SOS [6]) to rewrite extended expressions, in which additional information has been added to compute causal and weakly-causal dependencies. This information is also made visible by extending the labeling of transitions. Concurrency is just the complement of the weak-causal relation, and conflicts are defined by cycles in this relation. Capturing causality and concurrency by instrumenting the semantics rules is a difficult job. This is mainly due to the fact that these relationships are global and therefore difficult to locate on the syntactic forms. The solution is to keep information about the causal past in a context associated with each rule. We build the necessary links between different contexts during the execution of rules. The aim is that such instrumented semantics reproduces the standard behavior of the program while calculating the additional information needed to track concurrency, causality and conflicts between the events produced by the execution.

After this introduction, the article presents the contribution compared to the existing works. Section 3 presents the Orc language from the perspective of its core calculus and its operational semantics, illustrated using an example of orchestration of Web services. Section 4 presents our proposed instrumented semantics based on the construction of event structures, giving the concurrent semantics of Orc. This section sets out the formal correctness of the approach stating that this new semantics produces the same executions as the standard semantics. Before concluding the paper, Section 5 reuses the example of Section 3 to show the causal structure obtained from its execution in the instrumented semantics and how this can be used to find errors.

2 Related work

The need to dynamically trace the causal dependencies during the execution of the a program in order to monitor, detect errors or analyze performances is well recognized for concurrent applications. Causality, seen as a partial order [7], can be tracked in different ways. Some works are based on an instrumentation of either the underlying operating system or the source code. For example, vector

clocks have been widely used by the distributed algorithms community in the context of message-passing systems [8]. The context of Web service orchestrations is more complex as a language like Orc can generate unbounded concurrency patterns. To our knowledge, the only instrumentation made on programs is [9], based on Java byte-code. However, in the considered model, the only source of causality comes from variable accesses. The second approach is to change the semantics so that it produces causal information which leads to a *concurrent semantics*. The challenge is then to maintain a good form of equivalence with the original semantics. Several debugging techniques rely on this principle, especially for performing replay ([10] is a good example for a fragment of the Oz language). The most successful works in concurrent semantics were conducted on process algebra (e.g. pi-calculus [11]). Our contribution is in the same vein, but for the Orc language, in the complex context of wide-area computing. Other attempts of concurrent semantics for Orc based on event structures have already been published [12, 13]. They use an ad hoc connection of Petri net diagrams or Join calculus. It is not clear how this semantics can be implemented in practice at compile-time that transforms the source code into a concurrent model. An instrumented semantics solves this problem and allows to catch causal dependencies at runtime.

3 The Orc Programming Language

3.1 Core calculus

Orc is a full programming language, that looks like a functional language with many non-functional aspects to handle concurrency. The interested reader can refer to [14] concerning the ability of Orc to design large-scale distributed applications. The Orc programming language is designed over a process calculus: the Orc core calculus. All the conveniences offered in the full Orc language are derived from very few central concepts present in the calculus: sites and operators. Values such as booleans, numbers and strings, arithmetic and logic operators, as well as complex data types such as shared registers, are just external sites. Even choices are implemented through the use of sites `ift` and `iff`, that publish a signal if their argument is true or false respectively. Besides sites, four operators are provided by the calculus to orchestrate the execution. These operators describe the sequencing of actions ("$f >x> g$"), the launching of parallel threads ("$f|g$"), an original preemption operator ("pruning: $f <x< g$") and an alternative in case of no response ("otherwise: $f; g$"). The full syntax of the calculus is specified by the grammar given in Figure 1. From now on, we denote by Orc_s the set of the expressions allowed by this syntax. The expressions of the calculus that correspond to real Orc programs, denoted by the set Orc, are those that do not contain $?k$ and $\bot$ expressions.

There are two kinds of sites in Orc: the external ones, denoted V in the syntax, and the internal ones defined as an Orc expression with the syntax **def** $y(x) = f\#g$ where f is the body of the site and g is the remaining of the program in which y can be used as any site. For the sake of clarity, we consider

$$
\begin{array}{lll}
f, g, h \in \text{Expression} & ::= & p \| p(p) \| ?k \| f | g \| f >x> g \| f <x< g \| f; g \| D \# f \| \bot \\
D \in \text{Definition} & ::= & \textbf{def}\ y(x) = f \\
v \in \text{Orc Value} & ::= & V \| D \\
p \in \text{Parameter} & ::= & v \| \textbf{stop} \| x \\
w \in \text{Response} & ::= & NT(v) \| T(v) \| Neg \\
n \in \text{Hidden Label} & ::= & ?V_k(v) \| ?D \| h(\omega) \| h(!v) \\
l \in \text{Label} & ::= & !v \| n \| \omega
\end{array}
$$

Fig. 1. The syntax of the Orc core calculus.

in this work that the sites are curryfied, so they have exactly one argument. Site definitions are recursive, which allows the same expressivity as any functional language. Calls to external sites are strict, i.e. their arguments have to be bound before the site can be called, while an internal site can be called immediately, and its arguments are evaluated lazily. When an external site is called, it sends its responses to a placeholder $?k$. A response can be either a non-terminating value $NT(v)$ if further responses are expected, or a terminating value $T(v)$ if this is the last publication of the site, or Neg if the site terminates without publishing any value. In $f|g$, the parallel composition expresses pure concurrency; f and g are run in parallel, their events are interleaved and the expression stops when both f and g have terminated. Sequentiality can be expressed by the sequential operator, like in $f >x> g$, where the variable x can be used in g. Here, f is started first, and then a new instance of $g[v/x]$, where x is bound to v, is launched as a consequence of each publication of v. In $f <x< g$, the pruning operator is used to express preemption. The variable x can be used in f. Both f and g are started, but f is paused when it needs to evaluate x. When g publishes a value, it is bound to x in f, and g is stopped. Other events that could have been produced by g are preempted by the publication. For example, if g is supposed to publish two values a and b, only one will be selected and published in each execution. We say that these two events are in conflict. The pruning operator is left-associative: in $f <x< g <y< h$, f, g and h are started in parallel, the first publication of g is bound to x and the first publication of h is bound to y. The otherwise operator is used in $f; g$. In this expression, f is first started alone and g is started if and only if f stops without publishing any value. Finally, the **stop** symbol can be used by the programmer exactly like a site or a variable to denote a terminated program. **stop** still produces an event ω to notify its parent expression that it has terminated. It then evolves into $\bot$, the inert final expression. $?k$ and $\bot$ cannot be used directly.

3.2 Illustration

We now illustrate the use of Orc in Figure 3. This program defines the internal site `find_best(agencies, destination)` that computes the best offers proposed by the agencies listed in `agencies` for the destination given as a parameter. It publishes a unique value that is a pair composed of the best offer

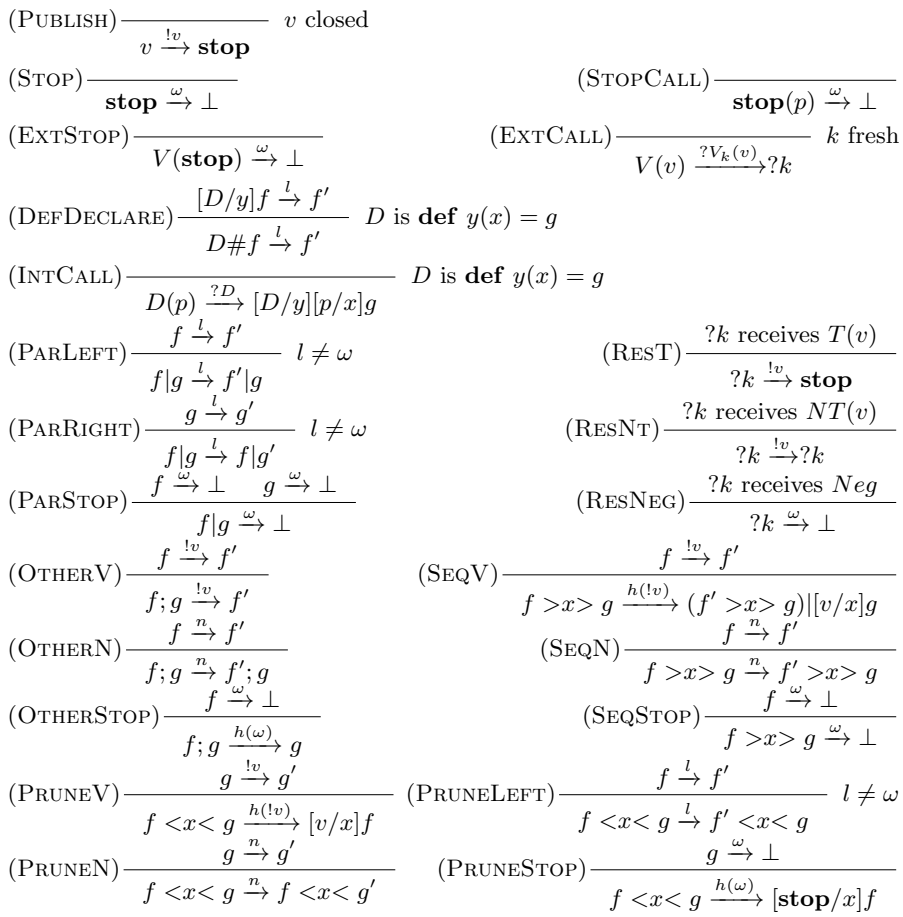

Fig. 2. The rules of the operational semantics.

augmented with additional information and the list of other offers sorted by price. The program is composed of three internal sites. It uses three shared objects, that are created in lines 15 to 17: the stack `offers` and the registers `best_offer` and `best_agency`. At line 16, a new register is created through a call to the site `Register()` and is bound to the variable `r`. It is then initialized to a default value: `r.write(null)` that can be seen as a shortcut for `r("write")>w>w(null)`, so the shared register is a site that can publish its accessors when it is called. As writing in a register does not publish any value, the otherwise operator is finally used to bound the value to `best_offer`. At line 11, the site `find_offers` can be started before the variables are created (left hand side of pruning operators). `each` publishes in parallel all the sites contained into the stack `agencies`, so all known agencies have to publish their offers. Each

```
1  def find_best(agencies, destination) =
2    def find_offers() =
3      each(agencies) > agency > agency(destination) > offer >
4      (offers.add((offer, agency)) |
5      (best_offer.read() > o > compare(o, offer) > b >
6      ift(b) > x > (best_agency.write(agency) | best_offer.write(offer))))          #
7    def extend_best() =
8      best_agency.read() > ba > best_offer.read() > bo > ba.exists(bo) > b >
9      (ift(b) > x > ba.get_info(bo) | iff(b) > x > alarm("inconsistent"))          #
10   def sort_offers(offers, best_offer) =
11     offers.sort(); best_offer.read() = offers.first() >b>
12     (ift(b) > x > offers | iff(b) > x > alarm("not best"))                       #
13   ((t <t< (find_offers() | timer(2000))) > t >
14   ((e_b, s_o) <e_b< extend_best() <s_o< sort_offers()))
15   <offers< Stack()
16   <best_offer< (Register() >r> r.write(null); r)
17   <best_agency< Register()                                                      #
```

Fig. 3. Identification of the best offers for a destination proposed by a pool of agencies.

time a new offer is found, it is added into `offer` and its price is compared to the current best known offer. The test is first evaluated and passed as an argument to `ift`. If true, the program publishes a signal and the registers can be updated. `find_offers` does not publish any value. In parallel with its call, we start a timer that publishes after 2 seconds a signal. The signal will halt this part of the program thanks to the pruning operator, and starts the line 14, thanks to the sequential operator. Line 14 calls both `extend_best` and `sort_offers` and publishes the result when both sites have published. The two sites call an external site either to sort the offers or to get extra information about the best offer, and they perform a test that raises an alarm if something wrong is detected.

Figure 4 shows a possible trace of the program of Figure 3. In this example, both alarms are due to inconsistencies in the shared registers. To avoid the alarm "inconsistent", it is necessary to write into `best_offer` and `best_agency` atomically, and to avoid the other alarm, the comparison with the current value of `best_offer` and its edition should be atomic. The event `best_offer.write(01)` is a cause for both alarms, but it is impossible to detect it in the sequential trace without any information about causality.

4 Instrumented Semantics

4.1 Method

SOS specifications take the form of a set of inference rules that define the valid transitions of a composite piece of syntax in terms of the transitions of its components. Rewriting transforms terms by executing a rule (it may be a nondeterministic transition in case of multiple alternatives). The successive transitions represent the program behavior. This may produce a sequence of values,

1. each([A1, A2])
2. timer(2000)
3. new_register()
4. new_register()
5. A1(D)
6. r.write(null)
7. best_offer.read()
8. new_stack()
9. offers.add(O1)
10. A2(D)
11. offers.add(O2)
12. compare(null, 01)
13. best_offer.read()
14. compare(null, 02)
15. ift(true)
16. ift(true)
17. best_offer.write(O2)
18. best_offer.write(O1)
19. best_agency.write(A1)
20. best_agency.write(A2)
21. best_agency.read()
22. best_offer.read()
23. A2.exists(O1)
24. iff(false)
25. ift(false)
26. alarm("inconsistent")
27. offers.sort()
28. best_offer.read()
29. offers.first()
30. =(O1, O2)
31. iff(false)
32. ift(false)
33. alarm("not best")

Fig. 4. A possible execution for the program in Figure 3 where agencies $= [A1, A2]$ and destination $= D$. Each agency publishes an offer $O1$ and $O2$ respectively. For the sake of space and clarity, we only show site calls in this execution.

that can be brought by the labeling of rules. Our approach is based on an instrumentation of the rules, that appends additional information to the labels in order to track the partial order of events. Actually, a label in the instrumented semantics is a tuple $e = (e_k, e_l, e_c, e_a)$, where e_k is an identifier taken in a countable set K, that is unique for the execution, e_l is a label similar to those of the standard semantics and e_c and e_a contain the finite sets of the identifiers of the causes and the weak causes of the event, respectively. Informally, an event e is a cause of e' if e always happens before e', regardless of the scheduling chosen by the system. Similarly, e is a weak cause of e' if e' can never happen after e, either because e is one of its causes or because e' preempts e.

In order to record the information concerning the past of an expression, we enrich the language with a new syntactic construction: $\langle f, c, a \rangle_L$ means that c and a are the causes of the Orc instrumented expression f. Thus, if f has c and a as causes and if it can evolve into f', this transition should also have c and a as causes. The index L expresses the kind of events that can activate the rule: $!v$ matches any publication, l stands for any label and ω means that c and a are only the causes of the termination of the program. We also consider that the external sites track causality themselves, as an internally-defined function would do. It makes sense as some sites (e.g. $+$) handle their calls independently, while others (e.g. shared registers, management library) induce more complex causality patterns between the calls. Hence, the responses we get include this additional information. The verification of these responses is not the subject here, and we suppose them to be correct by hypothesis.

Apart from the introduction of the instrumentation construction and the new information in the responses, the syntax of the instrumented expressions (Figure 5) is very similar to the regular one. The set of all the expressions allowed by this extended syntax is Orc_i. We can notice that every valid Orc program is also

$$
\begin{array}{lll}
f, g, h \in \text{Expression} & ::= & p \| p(p) \| ?k \| f | g \\
& & \| f >x> g \| f <x< g \| f; g \\
& & \| D \# f \| \bot \| \langle f, K, K \rangle_L \\
D \in \text{Definition} & ::= & \textbf{def}\ y(x) = f \\
v \in \text{Orc Value} & ::= & V \| D \\
p \in \text{Parameter} & ::= & v \| \textbf{stop} \| x \| \langle p, K, K \rangle_L \\
w \in \text{Response} & ::= & NT(v, K, K) \| T(v, K, K) \\
& & \| Neg(K, K) \\
n \in \text{Hidden Label} & ::= & ?V_k(v) \| ?D \| h(\omega) \| h(!v) \\
l \in \text{Label} & ::= & !v \| n \| \omega
\end{array}
$$

Fig. 5. The extended syntax of the instrumented semantics.

a valid instrumented expression, which means that the instrumented semantics can be applied without program transformation.

4.2 Labeled Asymmetric Event Structure

Labeled asymmetric event structures (LAES) [15] are natural objects to represent concurrent executions in a compact way.

Definition 1 (Labelled asymmetric event structure). *A* labelled asymmetric event structure *(LAES) is a tuple* $(E, L, \leq, \nearrow, \Lambda)$.

- E *is a set of* events,
- L *is a set of* labels,
- $\leq$, causality *is a partial order on* E,
- $\nearrow$, weak causality *is a binary relation on* E,
- $\Lambda : E \mapsto L$ *is the* labelling function,
- *each* $e \in E$ *has a finite* causal history $[e] = \{e' \in E | e' \leq e\}$,
- *for all events* $e < e' \in E, e \nearrow e'$, *where* $<$ *is the irreflexive restriction of* $\leq$,
- *for all* $e \in E$, $\nearrow \cap [e] \times [e]$, *the restriction of weak causality to the causal history of* e, *is acyclic.*

We also define an induced *conflict* relation $\#_a$ as the smallest set of finite parts of E such that: for $E' \subset E$ and $e_0, e_1, ..., e_n \in E$,

- if $e_0 \nearrow e_1 \nearrow \cdots \nearrow e_n \nearrow e_0$ then $\{e_0, e_1, \cdots, e_n\} \in \#_a$,
- if $E' \cup \{e_0\} \in \#_a$ and $e_0 \leq e_1$ then $E' \cup \{e_1\} \in \#_a$.

Informally, two events are in conflict if they cannot occur together in the same execution.

A LAES can be seen as a structure that encodes concisely several sequential executions; each of them being a linearization of the LAES.

Definition 2 (Linearization). *Let* $\mathcal{E} = (E, L, \leq, \nearrow, \Lambda)$ *be a LAES. A* finite linearization *of* $\mathcal{E}$ *is a word* $w = \Lambda(e_0) \dots \Lambda(e_n)$ *where the different* $e_i \in E$ *are distinct and such that:*

$$(\textsc{CauseYes})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{\langle f, c', a' \rangle_L \xrightarrow{k,l,c \cup c', a \cup a' \cup c'}_i \langle f', c', a' \rangle_L}\ l \in L$$

$$(\textsc{CauseNo})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{\langle f, c', a' \rangle_L \xrightarrow{k,l,c,a}_i \langle f', c', a' \rangle_L}\ l \notin L$$

Fig. 6. The semantics of the new operator $\langle f, c, a \rangle_L$ is defined by two additional rules.

- *it is left-closed for causality:*

$$\forall e \in E, \forall e' \in \{e_0, \ldots, e_n\}, e \leq e' \Rightarrow e \in \{e_0, \ldots, e_n\},$$

- *the weak causality is respected:*

$$\forall i, j \in \{0, \ldots, n\}, e_i \nearrow e_j \Rightarrow i < j.$$

We denote $Lin(\mathcal{E})$ as the set of all finite linearizations of $\mathcal{E}$.

Let $(E, L, \leq, \nearrow, \Lambda)$ be an asymmetric event structure and $e, e' \in E$ two events. We say that:

- e is a *cause* of e', if e happens before e' in all executions;
- e is a *weak cause* of e', if there is no execution in which e happens after e';
- e and e' are *concurrent*, denoted $e||e'$, if they can occur in either order. Formally, $e||e'$ if neither $e \nearrow e'$ nor $e' \nearrow e$.
- e is *preempted* by e', denoted $e \rightsquigarrow e'$, if e' can occur independently from e, but after that, e cannot occur anymore. Formally, $e \rightsquigarrow e'$ if $e \nearrow e'$ and $e \not\leq e'$.

4.3 Rules

Essentially, the instrumented semantics presented in Figure 7 decorates the rules of standard semantics, except that two rules are added (see Figure 6). The transition system defined by this instrumented semantics is denoted $\rightarrow_i$ and the sequential executions starting from a program f are contained in the set $[\![f]\!]_i$.

Informally, the expression $\langle f, c, a \rangle_L$ evolves exactly like f, but some causes and weak causes may be added to the event. For example, if $L = !v$ and f produces an internal event, that is not a publication, only the rule CauseNo can be applied, so the instrumentation will have no effect. On the other hand, if f publishes a value, the rule CauseYes applies and c and a are added to the causes and weak causes of the publication. Note that c is also added to the weak causes. This is to ensure that causality is always a special case of weak causality.

Let us now comment the most relevant instrumentations of the other rules. Let us consider rule SeqV. When a value is published, a new instance of the right hand side expression is created. All the events produced by this new expression need the former publication to *have occurred before* them, i.e. they are

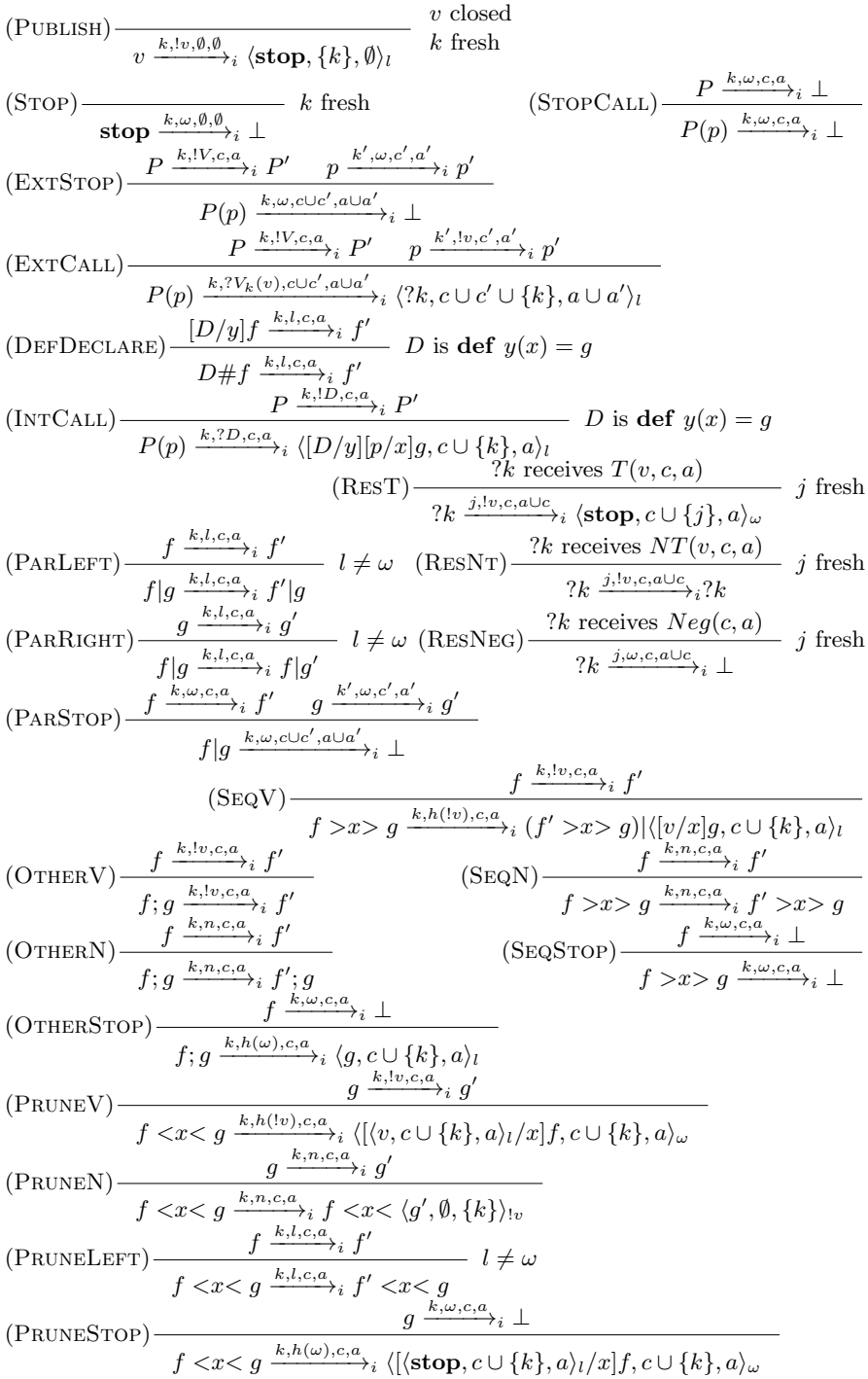

Fig. 7. The instrumented version of the rules of the operational semantics.

consequences of this publication. This is why the new expression is instrumented. Even if PRUNEV and SEQV are syntaxically very similar in their standard forms, the fact that both hand sides of the pruning operator are run in parallel makes them very different in terms of causality. In the expression $(1|x) <x< 2$, the second publication of 2 is a consequence of the first one, but not the publication of 1. This is why the instrumentation covers the occurrences of the newly bound variable. However, this is not sufficient. Consider the program $(\mathbf{stop} <x< 2); 3$. The publication of 3 must wait the end of the left hand side (i.e the publication of 2). However, this publication is useless, in the sense that no variable x can be bound to its value. To handle this case, we add an instrumentation to the whole expression that is only triggered when the expression stops. Finally, the rule PRUNEN is also interesting as it generates weak causality. Indeed, in the program $x <s< ((1 + 1)|3)$, the left hand side can call site $+$ and then publish 3, or publish 3 directly, but can never publish 3 and then call site $+$, because a publication preempts any other event. Of course, it could also wait for the answer of the site and then publish 2, which would preempt the publication of 3. This preemption relation is operated by an instrumentation that contains k as weak causes and that is triggered only in case of publication.

4.4 Concurrent executions

The equivalent of traces in the instrumented semantics are the *concurrent executions*, represented by LAES.

Definition 3 (Concurrent execution). *Let $\sigma = \sigma^1 \dots \sigma^n \in [\![f_0]\!]_i$, where for all i, $\sigma^i = (\sigma^i_k, \sigma^i_l, \sigma^i_c, \sigma^i_a)$. We define the* concurrent execution *of σ as the LAES:*

$$\overline{\overline{\sigma}} = (\{\sigma^1_k, \dots, \sigma^n_k\}, \{\sigma^1_l, \dots, \sigma^n_l\}, \leq, \nearrow, \Lambda)$$

where for all $i, j \in \{1, \dots, n\}$:

- $\sigma^i_k \leq \sigma^j_k$ *if* $\sigma^i_k \in \sigma^j_c$ *or* $i = j$,
- $\sigma^i_k \nearrow \sigma^j_k$ *if* $\sigma^i_k \in \sigma^j_a$,
- $\Lambda(\sigma^i_k) = \sigma^i_l$.

As the fields σ^i_c and σ^i_a only contain events that happened before σ^i in the sequential execution, both $\leq$ and $\nearrow$ are order relations and every event has a finite causal history. For the same reason, $\nearrow$ is acyclic. Moreover, it is easy to check that weak causality is more general than causality, so this definition actually corresponds to a real LAES.

We now state the main result: the behavior of a program is preserved by the instrumented semantics. It is established through two properties. The first one justifies the name of the instrumented semantics and the second one proves that the instrumentation is correct, i.e. that it does not define incorrect behaviors. Note that we do not give a complete proof of the two propositions for lack of space, but it can be found in [16].

Proposition 1 (Instrumentation). *The projections of the executions produced by the instrumented semantics on their labels correspond exactly to the executions of the standard semantics:*

$$\forall f \in Orc, \{\sigma_l^1 ... \sigma_l^n | \sigma \in [\![f]\!]_i\} = [\![f]\!].$$

In other words, it is always possible to instrument a standard execution to get a concurrent execution, and conversely we can get a standard execution from an instrumented execution by a simple projection.

The proof of this property is straightforward since both semantics contain similar rules. The only difficulty comes from the applications of CauseYes and CauseNo, that slightly modifie the structure of the derivation trees. All in all, both executions are similar.

Proposition 2 (Correctness). *The linearizations that can be inferred from an execution in the instrumented semantics are correct with respect to the standard semantics:*

$$\forall f \in Orc, \forall \sigma \in [\![f]\!]_i, Lin(\overline{\sigma}) \subset [\![f]\!].$$

This proof is much more complicated. Let us consider a linearization $L \in Lin(\overline{\sigma})$. To prove that $L \in [\![f]\!]$, we show that it is possible to progressively transform $\sigma_l^1 ... \sigma_l^n$ into L by applying a series of small steps that correspond either to the inversion of two consecutive concurrent events, to the preemption of an event by its successor or to a prefixation of the sequence. As $\sigma_l^1 ... \sigma_l^n \in [\![f]\!]$ and each step preserves this property, we get that $L \in [\![f]\!]$. The main difficulty concerns the proof of the correctness of the two first steps, as it requires a proof for all pairs of possible consecutive rules.

By introducing concurrency and preemption between events that were arbitrarily ordered by the standard semantics, the instrumented semantics gathers many sequential executions into one concurrent execution, which hugely reduces the number of different executions. However, all the events contained in a concurrent execution are also contained into a single sequential execution. Therefore, no instrumentation is able to capture conflict, as two conflictual events would never occur together. This is why completeness cannot be achieved with this approach.

5 Application

Let us reuse the example presented in Section 3 and the execution of Figure 4. Figure 8 shows the trace augmented with the causal information gained by the instrumented semantics.

Figure 9 shows the LAES in its graphical form. Events taht correspond to site calls are represented in circles and connected by three kinds of arrows. Direct causality is figured by solid and dashed arrows. Solid arrows represent program causality, that is specified by the instrumented semantics, while dashed arrows

(1, l_1, ∅, ∅)
(2, l_2, ∅, ∅)
(3, l_3, ∅, ∅)
(4, l_4, ∅, ∅)
(5, l_5, {1}, {1})
(6, l_6, {4}, {4})
(7, l_7, {1,4-6}, {1,4-6})
(8, l_8, ∅, ∅)
(9, l_9, {1,5,8}, {1,5,8})
(10, l_{10}, {1}, {1})
(11, l_{11}, {1,8,10}, {1,8,10})
(12, l_{12}, {1,4-7}, {1,4-7})
(13, l_{13}, {1,4,6,10}, {1,4,6,10})
(14, l_{14}, {1,4,6,10,13}, {1,4,6,10,13})
(15, l_{15}, {1,4-7,12}, {1,4-7,12})
(16, l_{16}, {1,4,6,10,13,14}, {1,4,6,10,13,14})
(17, l_{17}, {1,4,6,10,13,14,16}, {1,4,6,10,13,14,16})
(18, l_{18}, {1,4-7,12,15}, {1,4-7,12,15})
(19, l_{19}, {1,3-7,12,15}, {1,3-7,12,15})
(20, l_{20}, {1,3,4,6,10,13,14,16}, {1,3,4,6,10,13,14,16})
(21, l_{21}, {2}, {2, 1})
(22, l_{22}, {1-7,10,12-16,19-21}, {1-7,10,12-16,19-21})
(23, l_{23}, {1-7,10,12-19,22}, {1-7,10,12-19,22})
(24, l_{24}, {1-7,10,12-19,22,23}, {1-7,10,12-19,22,23})
(25, l_{25}, {1-7,10,12-19,22,23}, {1-7,10,12-19,22,23})
(26, l_{26}, {1-7,10,12-19,22-24}, {1-7,10,12-19,22-24})
(27, l_{27}, {2}, {2, 1})
(28, l_{28}, {1,2,5,9-11,27}, {1,2,5,9-11,27})
(29, l_{29}, {1,2,5,9-11,27}, {1,2,5,9-11,27})
(30, l_{30}, {1,2,4-7,9-18,27-29}, {1,2,4-7,9-18,27-29})
(31, l_{31}, {1,2,4-7,9-18,27-30}, {1,2,4-7,9-18,27-30})
(32, l_{32}, {1,2,4-7,9-18,27-30}, {1,2,4-7,9-18,27-30})
(33, l_{33}, {1,2,4-7,9-18,27-31}, {1,2,4-7,9-18,27-31})

Fig. 8. An execution augmented with causal information. Numbers refer to the events of Figure 4.

represent data causes, that are managed by the sites themselves. A *call* to a write on a site is a cause of the *publications* of the next read on this site, so the write is a cause of all the consequences of the read. Moreover, preemption is figured by dotted arrows. The call to `each()` — and all its consequences — is preempted by the publication made by `timer(2000)` — and all its consequences.

Root causes analysis. This execution concurrently raises two alarms. Let us consider their last common causes, i.e. the events that are causes of both alarms, and that are not causes of another such event. The alarms have two last common causes: `timer(2000)` and `best_offer.write(O1)`. The timer is not to blame here, as it has no causes and is just used as a starter for the program. Indeed, `best_offer.write(O1)` is the root cause for these two alarms. If this event did not exist, the value published by `best_offer.read()` would be `O2` for both calls, `A2.exists(O2)` and `=(O2, O2)` would be true and no alarm would be raised.

Detection of Race Conditions. We can see that the events `best_offer.write(O1)` and `best_offer.write(O2)` are concurrent, as well as `best_agency.write(O1)` and `best_agency.write(O2)`. In this context, these events can interleave so that `best_offer` and `best_agency` get inconsistent values.

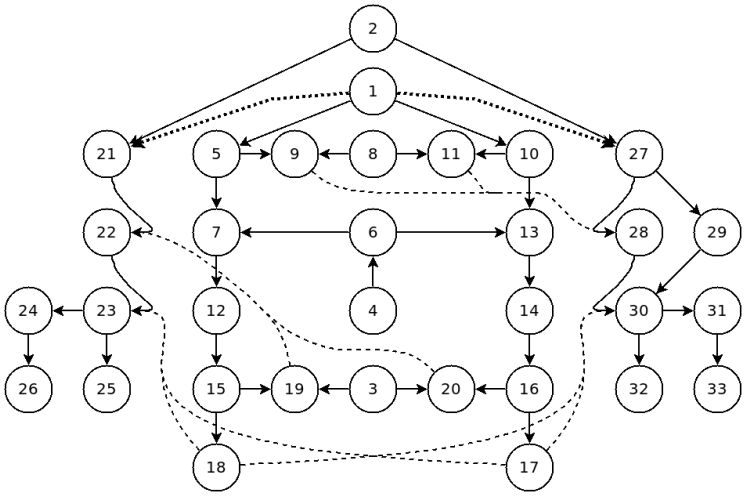

Fig. 9. Corresponding LAES in a graphical form.

6 Conclusion

We based our work on the Orc core calculus, as it is expressive enough to easily generate many situations found in distributed systems, such as causality, concurrency and preemption, and remains simple enough to be tractable in a formal work. Our contribution consists of an instrumentation of the standard structural operational semantics of Orc that tracks causality and weak causality at runtime to build LAES, well suited to represent concurrent executions. We think LAES are an interesting tool to access important properties of orchestrations. We illustrate this point on two questions: root cause analysis and detection of race conditions. Beyond the Orc language, we think that the article presents a general approach that can be used for other non-deterministic languages with concurrency operators. Based on this work, we think it is possible to produce the same information using a source to source transformation. Such a technique would be easier to implement, as it does not require to modify the execution engine of the language.

References

1. Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yarob Goland, Johannes Klein, Franck Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, Sanjiva Weerawarana. *Business Process Execution Language for Web Services.* Version 1.1, May 5, 2003. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf
2. David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. *ORC language.* http://orc.csres.utexas.edu

3. David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. *The ORC programming language.* In David Lee, Antonia Lopes, and Arnd Poetzsch-Heffter, editors, Formal Techniques for Distributed Systems, volume 5522 of Lecture Notes in Computer Science, pages 1-25. Springer, 2009.
4. David Kitchin, William Cook, and Jayadev Misra. *A language for task orchestration and its semantic properties.* In Christel Baier and Holger Hermanns, editors, CONCUR 2006, Concurrency Theory, volume 4137 of Lecture Notes in Computer Science, pages 477-491. Springer, 2006.
5. Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. *Foundations for Web services Orchestrations: functional and QoS aspects.* ISOLA 2006, 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 15-19 November 2006, Cyprus.
6. Gordon D. Plotkin. *The Origins of Structural Operational Semantics.* J. Log. Algebr. Program. 60-61:3-15, 2004.
7. Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, Volume 21 Issue 7, July 1978, pages 558-565.
8. Colin J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering.* Proc. of the 11th Australian Computer Science Conference (ACSC'88), pages 56-66, February 1988.
9. Grigore Roşu and Koushik Sen. *An instrumentation technique for online analysis of multithreaded programs.* In volume 19 of Concurrency and Computation: Practice and Experience, pages 311-325. Wiley Online Library, 2007.
10. Elena Giachino, Ivan Lanese, Claudio Antares Mezzina. *Causal-consistent Reversible Debugging* Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science Volume 8411, 2014, pages 370-384.
11. Michele Boreale, Davide Sangiorgi. *A Fully Abstract Semantics for Causality in the π-Calculus.* Acta Informaticae 35(5): 353-400 (1998)
12. Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar and Claude Jard. *Event Structure Semantics of ORC.* 4th International Workshop on Web Services and Formal Methods (WS-FM 2007), pages 154-168.
13. Roberto Bruni, Hernán Melgratti and Emilio Tuosto. *Translating Orc features into Petri nets and the Join calculus.* 3rd International Workshop on Web Services and Formal Methods (WS-FM 2006), pages 123-137.
14. Jayadev Misra, William Cook. *Computation orchestration: a basis for wide-area computing.* Journal of Software and Systems Modeling 6(1): 83-110 (2007)
15. Glynn Winskel. *Event structures.* Advances in Petri Nets. Springer Lecture Notes in Computer Science 255, 1987.
16. Matthieu Perrin, Claude Jard, Achour Mostéfaoui. *Building a concurrent operational semantics for the ORC language.* Technical report, LINA, Université de Nantes (2014) https://hal.archives-ouvertes.fr/hal-01101340v2