Tooling Offline Runtime Verification against Interaction Models : recognizing sliced behaviors using parameterized simulation

Erwan Mahe ⬤ *, **Boutheina Bannour** ⬤ *, **Christophe Gaston** ⬤ *, **Arnault Lapitre** ⬤ *, and **Pascale Le Gall** ⬤ †

*Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
†Université Paris-Saclay, CentraleSupélec, F-91192, Gif-sur-Yvette, France

ABSTRACT Offline runtime verification involves the static analysis of executions of a system against a specification. For distributed systems, it is generally not possible to characterize executions in the form of global traces, given the absence of a global clock. To account for this, we model executions as collections of local traces called *multi-traces*, with one local trace per group of co-localized actors that share a common clock. Due to the difficulty of synchronizing the start and end of the recordings of local traces, events may be missing at their beginning or end. Considering such *partially observed* multi-traces is challenging for runtime verification. To that end, we propose an algorithm that verifies the conformity of such traces against formal specifications called *Interactions* (akin to Message Sequence Charts). It relies on parameterized simulation to reconstitute unobserved behaviors.

KEYWORDS interaction, simulation, co-localization, shared clock, partial observation, multi-trace slice

1. Introduction

Runtime Verification (RV) ([Bartocci et al. 2018](#); [Sánchez et al. 2019](#)) refers to a group of techniques aiming at confronting observed system executions to formal references, specifying legal system executions, in order to identify non-conformance. Executions are observed via instrumentation and collected in *traces* consisting of sequences of atomic events. Such events often correspond to communication actions, consisting of emissions or receptions of messages observed at the system's interfaces under observation. Most approaches for RV can be described as either *offline* or *online*. In online approaches, events are processed on the fly whenever they are observed, while in offline approaches (abbrv. ORV) - which are the focus

of this paper -, traces are logged a priori to their analysis. Capturing executions of a Distributed System (DS) as traces is possible if it is observed via a unique interface deployed on a single machine. In practice, however, as a DS may be distributed across distinct machines, so is the instrumentation that observes its execution. Also, events observed on different and geographically distant computers cannot be easily temporally ordered as there is no common clock to label them with comparable dates. For these reasons, instead of a single global trace, one rather observes a set of local traces occurring on specific sub-systems or groups of *co-localized* (i.e. sharing a common clock) sub-systems. In such a situation, an execution is characterized as a structured collection of local traces, which we call a *multi-trace*.

While most RV techniques are based on formal references given in the form of automata ([Benharrat et al. 2017](#)) or temporal logic formulas ([Bauer & Falcone 2016](#)), we use an *interaction language*. Interactions are models whose most well-known instances are UML Sequence Diagrams (UML-SD) ([OMG 2017](#)) or Message Sequence Charts (MSC) ([ITU 2011](#)). Interactions specify the com-

munication flow between entities constituting a system. They are particularly adapted to specify DS behaviors, as DS are, by nature, composed of sub-systems interacting via message passing.

The graphical representation of interactions provides an intuitive vision of a DS's expected behaviors. Each sub-system is represented by a vertical line called a *lifeline* while message passing between sub-systems is represented by horizontal arrows drawn between the corresponding lifelines. As time flows from top to bottom, behaviors expected to occur on a given lifeline are sequences of emissions and receptions of messages that match the horizontal arrows entering or exiting the lifeline from top to bottom. More complex behaviors can be specified via operators drawn as annotated boxes.

In previous works (Mahe et al. 2022), we defined the semantics of interactions without the need for translations to other formalisms. In particular, an operational semantics in the style introduced by Plotkin for process algebras (Plotkin 2004) can be used to animate interaction models, explore their semantics, and define ORV algorithms (Mahe et al. 2020, 2021). This semantics is based on the key notion of *follow-up interaction*. Given an initial interaction i which specifies a set of expected behaviors, if a certain communication action a (either an emission or a reception of a message) inside i can immediately occur, then there exists a follow-up i', which we denote by $i \xrightarrow{a} i'$, such that i' specifies continuations of behaviors of i that start with the occurrence of a. Such atomic execution steps can be used to display graphically the semantics of an interaction in a tree-like structure. Being grounded by this small-step semantics, our approach is, to the best of our knowledge, the first tooled approach to offer interaction animation without going through translation mechanisms to intermediate formalisms like automata (Bannour et al. 2011), Petri-nets (Faria & Paiva 2016) or others as overviewed in (Mouakher et al. 2022).

The main contributions of this paper are extensions of the work in (Mahe et al. 2021, 2022) under two aspects:

(1) The observability constraints imposed by monitoring:

– We define a finer notion of multi-trace, whose component local traces are defined on groups of co-localized lifelines (e.g. that share a common clock) rather than on single lifelines. Analyzing those richer multi-traces allows taking advantage of the additional information provided by the existence of common clocks;
– We define a new ORV algorithm tolerant to the absence of synchronization related to both the beginning and the end of observation across distant monitors. To do so we rely on parameterized simulation for guessing missing/unobserved behaviors. With this new algorithm, logging on each co-localization can start and end independently and at any time when the system is in operation;

(2) Our specification formalism and tool implementation:

– We introduce a *coreg_r* operator for specifying behaviors that ought to be concurrent on specific sub-systems (the concurrent region r) while being weakly sequential on others. This new operator allows defining more expressive (w.r.t. the language from (Mahe et al. 2022)) specifications. It also simplifies the definition of the language, the weakly sequential *seq* and interleaving *par* operators being covered by its definition (being resp. equivalent to *coreg_∅* and *coreg_L* where L is the set of all lifelines).
– We also propose a reformulation of the operational semantics from (Mahe et al. 2022) which has the advantage of using fewer inductive predicates;
– We extend the tool implementation mentioned in (Mahe et al. 2021) to include our new contributions and present its interface in more details. This tool, called HIBOU, allows designing interactions, exploring their semantics, generating multi-traces and performing RV. We also propose an experimental evaluation of our simulation-based ORV approach using this tool.

The paper is organized as follows. In Sec.2 we introduce the notions of *multi-trace* for characterizing behaviors of DS and of *multi-trace slice* for characterizing partial observations of such behaviors. Then, in Sec.3 we define our language of *interactions* and its semantics in terms of multi-traces. Following those definitions, we introduce in Sec.4 a generic algorithm for verifying partially observed distributed behaviors (i.e. multi-trace slices) against formal specifications given in the form of interactions. This algorithm uses simulation steps in order to complete optimistically behaviors that might be missing from the slice (because it is not observed). It is generic in so far as the manner with which simulation is performed is parameterized by a certain criterion. After that, we propose one such criterion in Sec.5, apply the resulting algorithm on an example, discuss its advantages and limitations and present results from experiments. Related works and the position of our contribution are then discussed in Sec.6. Finally, in Sec.7, we present our tool implementation HIBOU.

2. Characterizing observed DS executions

2.1. Multi-traces to model executions

The asynchronous exchanges of messages are at the heart of the behaviors of Distributed Systems (DS). Those exchanges can be modeled using discrete communication actions (abbrv. as *actions*) corresponding to atomic emissions and receptions of messages. Those actions occur at the communication interfaces of specific sub-systems (those that emit and/or receive the corresponding messages) within the DS.

To formalize this, we describe the sub-systems constituting a DS using a set L of *lifelines*, and the messages that can transit through it using a set M of *messages*.

Elements of the set $\mathbb{A}$ of communication actions are :
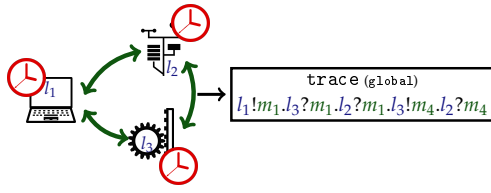
(a) 1 global clock ⇒ global trace
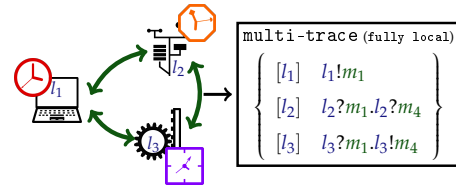
(b) 3 local clocks ⇒ "classical" multi-trace

(c) 2 shared clocks ⇒ "generalized" multi-trace

(d) partial observation ⇒ slice of multi-trace

Figure 1 Trace collection

- either of the form $l!m$, corresponding to the emission of the message m in M from the lifeline l in L
- or of the form $l?m$, corresponding to the reception of the message m in M by the lifeline l in L.

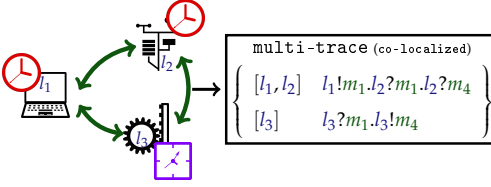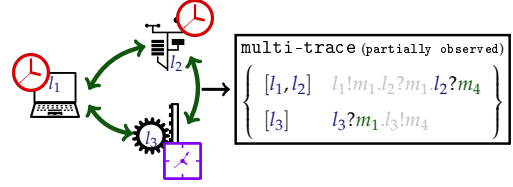For any action $a \in \mathbb{A}$ of the form $l!m$ or $l?m$, we denote by $\theta(a)$ the lifeline l on which a occurs.

An execution of a DS can be characterized by the actions that occurred in its span and by the order between their occurrences. Depending on the architecture of the DS our ability to reorder those actions may vary.

Fig.1a describes (on the left) a DS with three lifelines: l_1 (the computer icon), l_2 (the sensor icon) and l_3 (the gear icon) which all share the same global clock (as indicated by the drawn clocks). Thanks to the global clock, actions can be ordered globally, whichever is the sub-system on which they occur. Hence, an execution of the DS can be characterized by a global sequence of actions, which we call a *trace*. Traces are sequences of actions where ε represents the empty sequence and are concatenated using the "." operator. We denote by $\mathbb{T} = \mathbb{A}^*$ the set of all global traces[1]. The right side of Fig.1a describes an execution of our example DS in the form of a global trace:

$$l_1!m_1.l_3?m_1.l_2?m_1.l_3!m_4.l_2?m_4$$

This execution can be understood as follows: l_1 broadcasts message m_1 to both l_3 and l_2 and then l_3 sends m_4 to l_2.

Characterizing executions as global traces requires that all sub-systems share a common clock which we may call the global clock. However, in all generality, as the sub-systems of a DS can be distributed across distant machines, they may not share a common clock (Lamport 2019) and such a centralization and reordering of logging is not possible. Fig.1b describes a similar system as that of Fig.1a except that all three lifelines have different local clocks (as indicated by the drawn clocks). Let us suppose however,

[1] Given a set X, X^* denotes the set of all sequences with elements in X (this is the Kleene star notation).

that the same execution occurred in both cases. Then, because it is not possible to reorder actions occurring on distinct lifelines, instead of a global trace, the execution is rather characterized by a set of three local traces (one for each sub-system), which we call a *multi-trace*:

$$[l_1] \quad l_1!m_1$$
$$[l_2] \quad l_2?m_1.l_2?m_4$$
$$[l_3] \quad l_3?m_1.l_3!m_4$$

This notion of multi-trace can be found e.g. in (Benharrat et al. 2017; Mahe et al. 2021) as well as in (Attard & Francalanza 2017) (called partitioned traces) and (Nguyen et al. 2012; Bauer & Falcone 2016) (as sets of logs/local traces).

Still, it may be so that groups of sub-systems do share a common clock. We call those groups *co-localizations* (Pratt 1986). Fig.1c describes a variant of our example where lifelines l_1 and l_2 share a common clock (as indicated by the drawn clocks). In this case it is possible to order an action occurring on l_1 w.r.t. another occurring on l_2. As a result, the execution (the same as in the previous two cases) can be characterized by a *generalized* multi-trace where lifelines l_1 and l_2 constitute together a co-localization and where lifeline l_3 alone represents another co-localization. The multi-trace is then composed of two collected local traces, each representing a local order of actions on one of the two co-localizations.

$$[l_1, l_2] \quad l_1!m_1.l_2?m_1.l_2?m_4$$
$$[l_3] \qquad l_3?m_1.l_3!m_4$$

More formally, a *co-localization* c is defined by a subset of lifelines $c \subseteq L$. We introduce $\mathbb{A}_{|c} = \{a \in \mathbb{A} \mid \theta(a) \in c\}$ the set of actions occurring on a lifeline in c and $\mathbb{T}_{|c} = \mathbb{A}_{|c}^*$ the set of local traces defined on c.

For a set X, $\text{Part}(X)$ denotes the set of partitions of X, where a partition $C \in \text{Part}(X)$ is defined as a collection $C \subset \mathcal{P}(X)$ s.t. $\bigcup_{c \in C} c = X$ and $\forall\, (c, c') \in C, c \neq c' \Rightarrow c \cap c' = \emptyset$.

Definition 1 (Multi-traces). *Given a partition $C \in \text{Part}(L)$ of lifelines, we denote by $\mathbb{M}_C$ the set of multi-traces up to C. A multi-trace $\mu \in \mathbb{M}_C$ is defined as a tuple of traces, each defined over events occurring on a specific co-localization from C, hence $\mathbb{M}_C = \prod_{c \in C} \mathbb{T}_{|c}$.*

Given a multi-trace $\mu \in \mathbb{M}_C$ and given any $c \in C$, we denote by:

- $\mu_{|c}$ the local component of μ on c,
- for any $t \in \mathbb{T}_{|c}$, $\mu[t]_c$ the multi-trace μ in which t substitutes the c component, i.e. $\forall\, c' \in C \setminus \{c\}, (\mu[t]_c)_{|c'} = \mu_{|c'}$ and $(\mu[t]_c)_{|c} = t$.

We also extend the notation θ s.t. for any $a \in \mathbb{A}$ and $C \in \text{Part}(L)$, $\theta_C(a)$ designates the unique co-localization $c \in C$ on which a occurs i.e. s.t. $\theta(a) \in c$.

ε_C denotes the empty multi-trace s.t. $\forall\, c \in C, \varepsilon_{C|c} = \varepsilon$. We define a left concatenation operator for multi-traces as follows: for any action a and multi-trace μ, $a \mathbin{\vec{\odot}} \mu = \mu[a.\mu_{|\theta_C(a)}]_{\theta_C(a)}$ is the multi-trace obtained by prepending action a on the corresponding component of μ (i.e. $\mu_{|\theta_C(a)}$).

Finally, we denote by $C_t = \{L\}$ the trivial partition (in which all lifelines are co-localized) and by $C_d = \{\{l\} \mid l \in L\}$ the discrete partition (in which no two lifelines are co-localized).

Multi-traces defined up to C_t and C_d respectively correspond to the notions of global traces (Mahe et al. 2020) and multi-traces as defined in (Mahe et al. 2021). Co-localizations allow us to generalize and bridge the gap between those two notions. Hence, any RV approach that can handle those generalized multi-traces can also handle both global traces (trivial partition C_t) and classical multi-traces (discrete partition C_d) which are particular cases of generalized multi-traces.

As a side note, it is possible to define projections from coarser multi-traces to finer multi-traces. For any set X, a partition $C' \in \text{Part}(X)$ is a refinement of a partition $C \in \text{Part}(X)$, denoted by $C' \leq C$, if for any $c' \in C'$, there exists $c \in C$ s.t. $c' \subset c$. For instance, the multi-traces from Fig.1c and Fig.1b can be obtained by projecting that from Fig.1a because $\{\{l_1, l_2\}, \{l_3\}\} \leq \{\{l_1, l_2, l_3\}\}$ and $\{\{l_1\}, \{l_2\}, \{l_3\}\} \leq \{\{l_1, l_2, l_3\}\}$.

2.2. Slices to model partially observed executions

As we have seen, the example execution from Fig.1a is characterized by the global trace :

$$t = l_1!m_1.l_3?m_1.l_2?m_1.l_3!m_4.l_2?m_4$$

With the aim of performing Offline Runtime Verification we would then analyze this trace w.r.t. a certain behavioral model. However, in practice, the instrumentation that is used to collect this trace (from an observation of the execution that occurred) is not always perfect. If we suppose

that this trace t characterizes the entirety of the execution, then, depending on the quality of the instrumentation, the collected trace t' may be a suffix of t (if the observation started too late), a prefix of t (it it ended too early) or a sub-word of t (if both).

This remark can equally be applied in the absence of a global clock. In the case of Fig.1c, l_1 and l_2 share a common clock while l_3 has its own local clock. Thus, the instrumentation must have at least two distinct observers to log actions on those two co-localizations. In ideal conditions, the observation of this execution would yield the following multi-trace (see also Fig.1c) defined over $C = \{\{l_1, l_2\}, \{l_3\}\}$:

$$[l_1, l_2] \quad l_1!m_1.l_2?m_1.l_2?m_4$$
$$[l_3] \quad l_3?m_1.l_3!m_4$$

In practice however, it may be difficult to synchronize the periods of observation between distant observers. Hence it may be so that some actions are missing at the beginning and/or at the end of local components of the multi-trace (as compared with the multi-trace that would have been observed in ideal conditions of observation).

This is illustrated on Fig.1d. Here, we suppose that observation on component $\{l_1, l_2\}$ started too late, leading to $l_1!m_1$ and $l_2?m_1$ not being observed while that on component $\{l_3\}$ ceased too early, leading to $l_3!m_4$ not being observed. With the convention that the missing actions are greyed-out, this leads to the following partial multi-trace:

$$[l_1, l_2] \quad l_1!m_1.l_2?m_1.l_2?m_4$$
$$[l_3] \quad l_3?m_1.l_3!m_4$$

This notion of partial observation (i.e. some elements may be missing at the beginning or at the end of traces composing the multi-trace) corresponds to considering *slices of multi-traces*, as defined in Def.2.

Definition 2 (Slices). *For any traces $t, t' \in \mathbb{T}_{|c}$, we say that t' is a slice of t iff there exists t_+ and $t_- \in \mathbb{T}_{|c}$ s.t. $t = t_-.t'.t_+$ and we denote by $\widetilde{t}$ the set of slices of t.*
For any multi-traces $\mu, \mu' \in \mathbb{M}_C$, we say that μ' is a slice of μ iff for all $c \in C$, $\mu'_{|c} \in \widetilde{\mu_{|c}}$ and $\widetilde{\mu}$ denotes the set of slices of μ.

In our example, if we denote by μ_0 the multi-trace from Fig.1c (observed in ideal conditions) then the multi-trace described on Fig.1d (collected in conditions of partial observation) is a slice of μ_0 i.e. $\mu'_0 \in \widetilde{\mu_0}$.

3. Interactions and their semantics

3.1. Syntax

Formal behavioral models enable users to

1. specify systems which may exhibit infinitely many distinct behaviors as finite expressions and

2. automate Verification and Validation processes such as RV.

The challenge of modeling and adapting RV to complex DS requires rich and intuitive formalisms. Interactions (Mahe et al. 2022) are well-suited to specify distributed behaviors thanks to their intuitive graphical representation (in the fashion of UML-Sequence Diagrams (Micskei & Waeselynck 2011)) while at the same time being formal models on which RV techniques can be applied (Mahe et al. 2020, 2021). In that spirit, an interaction is both described as a syntactic term which takes the form of a binary tree (see Fig.2b) and visualized as a diagram (see Fig.2a) which may be familiar to many software engineers thanks to the wide use of UML-SD and MSC. Fig.2 depicts an example of interaction defined on the lifelines l_1, l_2 and l_3 with messages m_1, m_2, m_3 and m_4. It is represented as a diagram on the left (Fig.2a), and as a term on the right (Fig.2b).
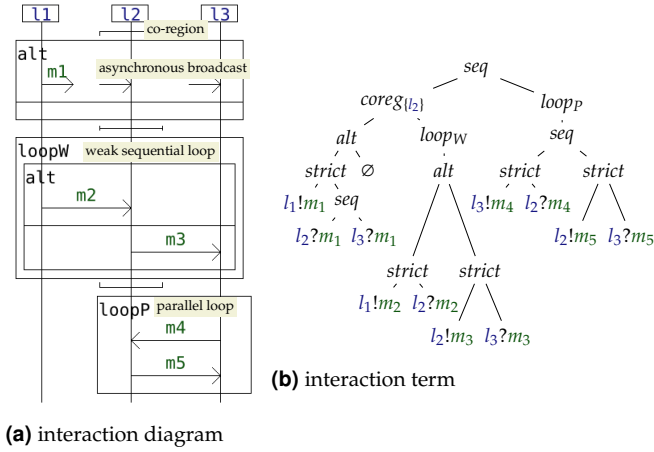

(b) interaction term

(a) interaction diagram

Figure 2 An example of an interaction

Interaction models correspond to expressions built over:

– the empty interaction, denoted by $\varnothing$, with the empty multi-trace ε_C as the only accepted one
– and actions a (of the form $l!m$ or $l?m$) with a multi-trace reduced to a single action as the only accepted one (i.e. the multi-trace $a \overset{\rightarrow}{\odot} \varepsilon_C$).

We then use operators to compose interactions into more complex expressions. Let us consider two interactions i_1 and i_2:

– *alt* stands for alternative and a behavior of $alt(i_1, i_2)$ is either a behavior of i_1 or one of i_2 according to a non-deterministic and exclusive choice between the two alternatives.
– *strict* stands for strict sequencing and a behavior of $strict(i_1, i_2)$ is such that a behavior from i_1 must be entirely expressed before any action from i_2 can occur;
– *coreg* stands for concurrent region and corresponds to a family of operators $(coreg_r)_{r \in \mathcal{P}(L)}$. For a given subset $r \subseteq L$ of lifelines, $coreg_r(i_1, i_2)$ specifies behaviors composed from behaviors of i_1 and i_2.

(a) In the first case $r = L$, actions occurring in i_1 and i_2 can occur in any order in behaviors expressed by $coreg_r(i_1, i_2)$. This definition coincides with a classical interleaving (Knapp & Mossakowski 2017; Mahe et al. 2022) or orthocurrence (Pratt 1986) operator. As such we denote by *par* the corresponding derivable construct.
(b) In the second case $r = \emptyset$, interleaving is only possible between actions that occur on different lifelines, i.e. the behaviors of $coreg_\emptyset(i_1, i_2)$ are defined as with the *strict* operator for actions occurring on the same lifeline (whatever it may be) and as with the *par* operator for actions occurring on different lifelines. This definition coincides with weak sequencing (Knapp & Mossakowski 2017; Mahe et al. 2022) which is a key operator for sequence diagrams. As such we denote by *seq* the corresponding derivable construct.
(c) In the last case $r \notin \{\emptyset, L\}$, $coreg_r$ behaves as *par* on r and as *seq* on $L \setminus r$.

The *coreg* operator is new w.r.t. the language from (Mahe et al. 2022). This operator is inspired by the co-regions of UML-SD (OMG 2017), also found in some papers on MSCs (Katoen & Lambert 1998). The *coreg* construct allows certain patterns of communications that would be difficult to model ergonomically otherwise. For instance, on Fig.2 we have that (1) l_1 has to emit m_1 (if it ever does) before it can emit m_2 (if it ever does) and (2) l_2 can receive m_1 or m_2 in any order. To specify this, we use a co-region on lifeline l_2. This could not have been done using a *seq* (as it would forbid m_2 to be received before m_1) or a *par* (as it would allow m_2 to be emitted before m_1).

seq and *par* are not primitive operators as they can be derived from *coreg*. However, because those two operators are familiar to users of sequence-diagram-like models and widely used, we keep them to denote the corresponding *coreg* variants.

strict and *coreg* are binary scheduling operators i.e. they can be used to schedule behaviors w.r.t. one another. As a result, they can be used for defining repetition operators in the same manner as concatenation can be used to define the Kleene star for regular expressions. In (Mahe et al. 2022) we have defined $loop_S$, $loop_W$, $loop_P$ as repetition operators using resp. *strict*, *seq* and *par*. In the same fashion we can define a family $(loop_{C_r})_{r \in \mathcal{P}(L)}$ of repetition operators. In contrast to MSC and UML-SD, which only have a single loop construct, those loops enable us to specify a variety of behaviors.

$loop_S$ is a strict sequential loop meaning that any instance of the repeated behavior must be entirely executed (globally) before any other instance of the behavior might be started.

$loop_C$, used as $loop_{C_r}(i)$ with $r \subseteq L$ is a repetition using a $coreg_r$ operator. It is a middleground between:

– $loop_W = loop_{C_\emptyset}$ which corresponds to repetitions using the weak sequential operator. Several instances of the repeated behavior might exist at the same time

because there is no synchronization between lifelines as for the beginning and end of the executions of those instances. Moreover, with $loop_W$, it might be so that the first action that is executed does not come from the first instance of the loop. For instance, in the example from Fig.2, after a first occurrence of $l_1!m_2$, lifeline l_2 may emit m_3 several times before receiving the m_2 initially sent by l_1.

- and $loop_P = loop_{C_L}$, which is more akin to the bang operator of pi-calculus (Parrow 2001) and signifies the parallel composition of an arbitrary number of instances of the same behavior. It can be used to model services of which, at any given time, many instances may run in parallel.

When modeling DS, communications between subsystems can be defined up to a certain communication medium. In formalisms based on communicating automata, this often takes the form of buffers which assume certain policies (FIFO, bag, etc.) (Engels et al. 2002). In our case, loops used in combination with asynchronous message passing, may be used to abstract away those communication media. For instance, while $loop_W(strict(l_1!m, l_2?m))$ corresponds to having a FIFO buffer receiving messages from l_1 on l_2, in contrast, by using $loop_P(strict(l_1!m, l_2?m))$ we rather have a bag buffer in so far as instances of message m can be received in any order.

Definition 3. *The set $\mathbb{I}$ of interactions is the least term set s.t.:*
− ∅ and actions in $\mathbb{A}$ belong to $\mathbb{I}$
− for any $i_1, i_2 \in \mathbb{I}^2$ and any $r \subseteq L$:
* − ∀ $f \in \{strict, alt, coreg_r\}$, $f(i_1, i_2) \in \mathbb{I}$*
* − ∀ $k \in \{S, C_r\}$, $loop_k(i_1) \in \mathbb{I}$*

In Def.3, we formalize our interaction language. Let us keep in mind that the *seq*, *par*, $loop_W$ and $loop_P$ constructs are derivable from *coreg* and $loop_C$.

Via their recursive definition, interactions have a tree-like structure, as illustrated on Fig.2b. Those trees are binary-trees and we can pinpoint unambiguously each sub-tree via its position as a word $p \in \{1, 2\}^*$ (with ε the empty position which designates the root node). More precisely, 1 (resp. 2) allows access to the left direct sub-interaction or the unique direct sub-interaction (resp. the right direct sub-interaction). For any interaction i, $pos(i)$ designates the set of its positions, and, for any $p \in pos(i)$, $i_{|p}$ designates the sub-interaction at position p. For example, for the interaction $i = seq(alt(l_1!m_1, l_2?m_2), l_1!m_3)$, $i_{|1}$ is the interaction $alt(l_1!m_1, l_2?m_2)$, $i_{|12}$ is the interaction $l_2?m_2$ and $\{\varepsilon, 1, 2, 11, 12\}$ is the set of positions of i.

3.2. Semantics

Given a partition $C \in Part(L)$ defining co-localizations, each interaction $i \in \mathbb{I}$ characterizes a (potentially infinite) set $\sigma_C(i)$ of multi-traces according to C. This semantics can be defined in an operational-style using either inductive rules (in the style of Plotkin (Plotkin 2004)) as in (Mahe et

al. 2022) or through the definition of an execution function (in the style of functional programming languages) as in (Mahe et al. 2020, 2021). In the following we propose a reworked (w.r.t. that of (Mahe et al. 2022)) operational-style formulation which includes the coregion operator *coreg* and involves fewer inductive predicates.

Interactions provide detailed control structures as for the occurrences and orders of actions, going beyond the simple linear order of events made available by multitraces. An operational-style semantics defines accepted behaviors via concatenations of actions a which occurrences are associated to term transformations of the form

$$i \xrightarrow{a@p} i'$$

Here interaction i' specifies all the continuations of the behaviors specified by i which start with the occurrence of action $a \in \mathbb{A}$, which, by construction, is a sub-term of i at a certain position $p \in pos(i)$.

In order to ensure that the follow-up interaction i' specifies the right order of actions following the first selected action (a at position p) according to the interaction i, the execution relation $\rightarrow$ performs transformations on the initial term i so as to obtain i'. Those transformations may include pruning operations (related to the notion of "permission" in (Mauw & Reniers 1997)) which clean the term with regard to the lifeline of the action which is executed.
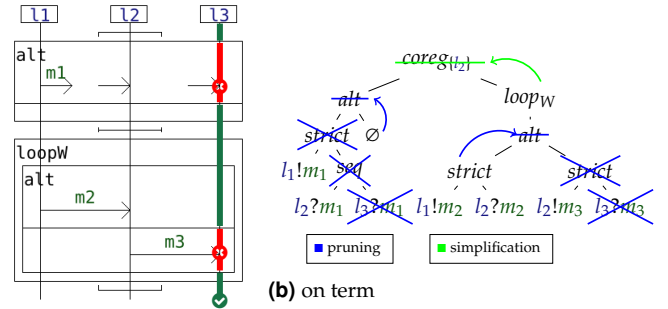
(a) on diagram

(b) on term

Figure 3 Pruning an interaction

In particular, the mechanism of pruning enters into play for handling weak sequencing. For example, let us consider executing $l_3!m_4$ in the interaction from Fig.2. If $l_3!m_4$ is the first action to occur then, so as to respect the top to bottom order of the diagram (i.e. weak sequencing), this means that neither $l_3?m_1$ nor $l_3?m_3$ can occur. Indeed, they appear above $l_3!m_4$ along the lifeline l_3 in Fig.2, more precisely they are scheduled with weak sequencing w.r.t. to it and must precede it. Hence, if those actions were to occur they would have done so before $l_3!m_4$. As a result, both actions must be eliminated or, in other words, to better conform to our vocabulary, pruned from the follow-up interaction. This is possible because they are within alternatives and loops. The general idea is to transform the sub-terms preceding (i.e. with sequencing) the action that

is executed (here $l_3!m_4$) in such a way as to eliminate from these sub-terms actions that involve the lifeline on which this executed action occurs (here l_3). In this process, which we call *pruning*, pertinent actions are eliminated and, from the bottom up, the *pruned* sub-terms are reconstructed so as to keep all the behaviors that do not involve a certain lifeline (here l_3).

This process of pruning is illustrated on Fig.3. The interaction term with $coreg_{\{l_2\}}$ as root in Fig.3b is simplified with two purposes: eliminate all traces with an action on lifeline l_3 and preserve all other accepted traces. As the sub-interaction $strict(l_1!m_1, seq(l_2?m_1, l_3?m_1))$ at position 1 only accepts traces containing the action $l_3?m_1$, the first alternative of the *alt* operator (position 11) is no longer allowed and the sub-interaction with *alt* as top operator is reduced to its second alternative, here $\varnothing$ (position 12), which by definition accepts only the empty trace and consequently avoids lifeline l_3. As can be seen from Fig.3a, the process of pruning an interaction is a local transformation guided by the lifelines to be avoided.

In Def.4 below, we define pruning w.r.t. a subset $L' \subseteq L$ of lifelines. The two pruning relations $\xmapsto{L'}$ and $\not\xmapsto{L'}$ are defined inductively on the term structure of interactions. For any interactions i and i':

- $i \xmapsto{L'} i'$ signifies that i' is an interaction which specifies exactly all the behaviors specified by i that do not involve any action occurring on a lifeline of L'.
- $i \not\xmapsto{L'}$ signifies that it is impossible to find such an interaction i' because all the behaviors specified by i involve at least one action occurring on a lifeline of L'.

Definition 4 (Pruning). *The pruning relations $\xmapsto{} \subset \mathbb{I} \times \mathcal{P}(L) \times \mathbb{I}$ and $\not\xmapsto{} \subset \mathbb{I} \times \mathcal{P}(L)$ are s.t. for any $L' \subseteq L$, any $f \in \{strict\} \cup \bigcup_{r \subseteq L}\{coreg_r\}$ and $k \in \{S\} \cup \bigcup_{r \subseteq L}\{C_r\}$:*

$$\frac{}{\varnothing \xmapsto{L'} \varnothing} \qquad \frac{}{a \xmapsto{L'} a} \theta(a) \notin L' \qquad \frac{}{a \not\xmapsto{L'}} \theta(a) \in L'$$

$$\frac{i_1 \xmapsto{L'} i'_1 \quad i_2 \xmapsto{L'} i'_2}{alt(i_1, i_2) \xmapsto{L'} alt(i'_1, i'_2)} \qquad \frac{i_1 \not\xmapsto{L'} \quad i_2 \not\xmapsto{L'}}{alt(i_1, i_2) \not\xmapsto{L'}}$$

$$\frac{i_j \xmapsto{L'} i'_j \quad i_{j'} \not\xmapsto{L'}}{alt(i_1, i_2) \xmapsto{L'} i'_j} \{j, j'\} = \{1, 2\}$$

$$\frac{i_1 \xmapsto{L'} i'_1 \quad i_2 \xmapsto{L'} i'_2}{f(i_1, i_2) \xmapsto{L'} f(i'_1, i'_2)} \qquad \frac{i_j \not\xmapsto{L'}}{f(i_1, i_2) \not\xmapsto{L'}} j \in \{1, 2\}$$

$$\frac{i_1 \xmapsto{L'} i'_1}{loop_k(i_1) \xmapsto{L'} loop_k(i'_1)} \qquad \frac{i_1 \not\xmapsto{L'}}{loop_k(i_1) \xmapsto{L'} \varnothing}$$

The pruning relations are defined inductively in the style of Plotkin (Plotkin 2004):

- the empty interaction $\varnothing$ can always be pruned into

$\varnothing$ w.r.t. any subset $L' \subseteq L$ of lifelines (i.e. $\varnothing \xmapsto{L'} \varnothing$) because it expresses no action occurring on L'

- for any action a, we have $a \xmapsto{L'} a$ if $\theta(a) \notin L'$ and $a \not\xmapsto{L'}$ otherwise because a must be expressed
- having $alt(i_1, i_2) \not\xmapsto{L'}$ (resp. $strict(i_1, i_2) \not\xmapsto{L'}$) requires that both (resp. one of the two) $i_1 \not\xmapsto{L'}$ and $i_2 \not\xmapsto{L'}$ hold
- all other cases are handled similarly.

We define the execution relation $i \xrightarrow{a@p} i'$ - which makes interactions executable - in the same way as the pruning predicates. Executing an atomic action $a \in \mathbb{A}$ simply consists in replacing it with the empty interaction $\varnothing$ because once action a is expressed, nothing remains to be expressed. If an action can be expressed within a branch of an alternative, then it can also be expressed from the alternative itself but its expression forces the choice of the branch on which it occurs to be made. An action which can be expressed on the left branch of a scheduling operator (*strict* or any $coreg_r$ which implies also *seq* and *par*) can always be expressed from the scheduling operator itself and what remains to be expressed is the scheduling of what remains of the left branch w.r.t. the initial right branch.

Defining $i \xrightarrow{a@p} i'$ for executing actions from the right branch of i can be more challenging. Indeed, it is not always possible to express an action on the right branch of a scheduling operator, and, if it is possible, then it often requires pruning the left branch so as to remove inconsistencies in the follow-up interaction.

When an action is expressed inside a loop, we have in the follow-up a certain scheduling of what remains to be executed of the instance of the sub-behavior w.r.t. the initial loop (which, serving as a specification of the repeatable instance, remains the same). Due to the peculiarities of weak sequencing (as evoked in (Mahe et al. 2022)), in particular to the fact that the first action that is executed does not necessarily come from the first instance of the loop (as ordered by the operator which schedule different instances of the loop), the rule for $loop_{C_r}$ is somewhat more complex. It involves scheduling a pruned version of the initial loop before the remainder of the executed instance.

Definition 5 (Execution). *The execution relation $\to \subset \mathbb{I} \times (\mathbb{A} \times \{1, 2\}^*) \times \mathbb{I}$ is defined as follows:*

$$\frac{}{a \xrightarrow{a@\varepsilon} \varnothing} \qquad \frac{i_1 \xrightarrow{a@p} i'_1}{alt(i_1, i_2) \xrightarrow{a@1.p} i'_1} \qquad \frac{i_2 \xrightarrow{a@p} i'_2}{alt(i_1, i_2) \xrightarrow{a@2.p} i'_2}$$

$$\frac{i_1 \xrightarrow{a@p} i'_1}{strict(i_1, i_2) \xrightarrow{a@1.p} strict(i'_1, i_2)} \qquad \frac{i_1 \xrightarrow{a@p} i'_1}{coreg_r(i_1, i_2) \xrightarrow{a@1.p} coreg_r(i'_1, i_2)}$$

$$\frac{i_1 \xmapsto{L} \varnothing \quad i_2 \xrightarrow{a@p} i'_2}{strict(i_1, i_2) \xrightarrow{a@2.p} i'_2} \qquad \frac{i_1 \xmapsto{\{\theta(a)\}\backslash r} i'_1 \quad i_2 \xrightarrow{a@p} i'_2}{coreg_r(i_1, i_2) \xrightarrow{a@2.p} coreg_r(i'_1, i'_2)}$$

$$\dfrac{i_1 \xrightarrow{a@p} i_1'}{loop_S(i_1) \xrightarrow{a@1.p} strict(i_1', loop_S(i_1))}$$

$$\dfrac{i_1 \xrightarrow{a@p} i_1' \qquad loop_{C_r}(i_1) \xLeftrightarrow{\{\theta(a)\}\backslash r} i'}{loop_{C_r}(i_1) \xrightarrow{a@1.p} coreg_r(i', coreg_r(i_1', loop_{C_r}(i_1)))}$$

In (Mahe et al. 2022), the reader can find detailed explanations on the rules concerning most operators (*strict*, *par*, *alt*, *seq* and *loop*). In addition, Def.5 includes the use of positions which is not the case in (Mahe et al. 2022). Using those decorations @p comes at no cost given the inductive nature of →'s definition. Moreover, it removes all ambiguities related to having several executable occurrences of the same action (i.e. $i \xrightarrow{a@p_1} i_1$ and $i \xrightarrow{a@p_2} i_2$ with $p_1 \neq p_2$ and $i_1 \neq i_2$).
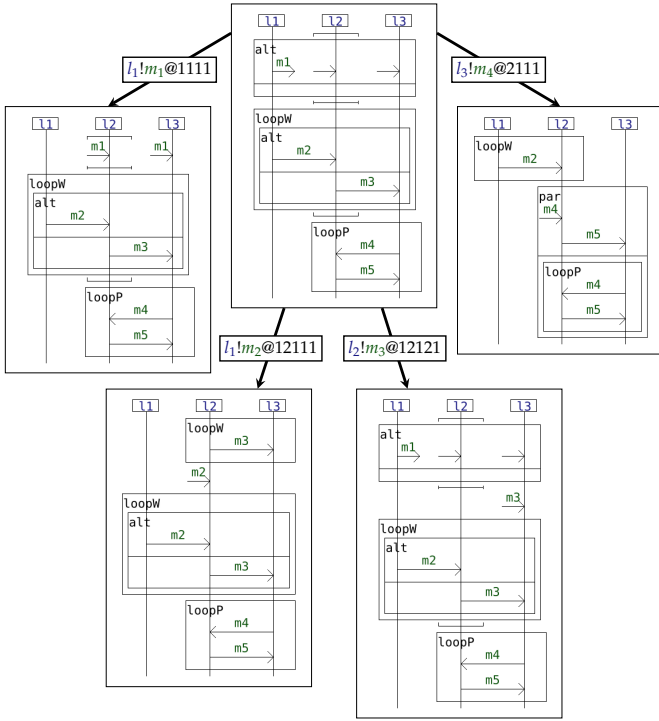

Figure 4 Follow-ups of the interaction from Fig.2

Fig.4 illustrates the use of the execution relation on the example from Fig.2. Four distinct actions can be immediately executed, leading to four follow-up interactions.

This process of computing follow-up interactions can be repeated recursively so that we obtain a tree which root is the initial interaction i_0. This tree, called an *execution tree* represents the semantics is i_0 i.e. its set of accepted behaviors. behaviors expressed by i_0 can indeed be observed via the succession of the actions that are executed on any path of the tree starting from i_0 and ending with an interaction that can express the empty behavior.

Finally, the set of multi-traces $\sigma_C(i_0)$ accepted by an interaction i_0 can be built using the execution relation →.

A multi-trace μ belongs to $\sigma_C(i_0)$ iff it can be written as:

$$\mu = a_1 \vec{\odot} a_2 \vec{\odot} \ldots \vec{\odot} a_n \vec{\odot} \varepsilon_C$$

and if there exist n interactions $i_1, \ldots, i_n$ s.t.:

- $\forall\, j \in \{0, 1, \ldots, n-1\}$, $i_j \xrightarrow{a_{j+1}@p_{j+1}} i_{j+1}$
- and $\varepsilon_C \in \sigma_C(i_n)$.

This last point (i.e. whether or not $\varepsilon_C \in \sigma_C(i_n)$) can be determined statically using the pruning relation. Indeed, if there exists i_n' such that $i_n \xLeftrightarrow{L} i_n'$ then this means that i_n has at least a behavior which does not involve any action that occurs on lifelines of L. L being the set of all lifelines, this behavior can only correspond to the empty multi-trace ε_C and hence $\varepsilon_C \in \sigma_C(i_n)$.

Definition 6. *Let $C \in Part(L)$ and let $i \in \mathbb{I}$.*
The semantics $\sigma_C(i)$ of i is the least subset of $\mathbb{M}_C$ s.t.:

$$\dfrac{i \xLeftrightarrow{L} i'}{\varepsilon_C \in \sigma_C(i)} \qquad \dfrac{\mu \in \sigma_C(i') \qquad i \xrightarrow{a@p} i'}{a \vec{\odot} \mu \in \sigma_C(i)}$$
$$\text{with } \mu \in \mathbb{M}_C, a \in \mathbb{A}, p \in \{1,2\}^* \text{ and } i' \in \mathbb{I}.$$

3.3. Soundness of the operational semantics

In (Mahe et al. 2022) we have given a denotational semantics for the interaction language without the *coreg* and *loop_C* operators. The semantics is based on the use of composition and algebraic operators as in (Knapp & Mossakowski 2017). In (Mahe et al. 2022), we used the ∥ (interleaving) and ⪥ (weak sequencing) operators on sets of traces. In order to include *coreg* we have to define a new operator on sets of traces as follows. The first step is to define a conditional conflict predicate $t \,⪥|_r\, l$ meaning that the trace t contains an action on a lifeline $l \notin r$:

$$\varepsilon \,⪥|_r\, l = \bot$$
$$(a.t) \,⪥|_r\, l = ((\theta(a) = l) \wedge (l \notin r)) \vee (t \,⪥|_r\, l)$$

If $t \,⪥|_r\, l = \top$, we say that the trace t has conflicts w.r.t. the lifeline l in the region covered by $L \setminus r$ where $r \subseteq L$ is the concurrent region. By overloading the symbol $⪥|_r$, the set $t_1 \,⪥|_r\, t_2$ of conditional sequencing of t_1 and t_2 is defined by:

$$\varepsilon \,⪥|_r\, t_2 = \{t_2\}$$
$$t_1 \,⪥|_r\, \varepsilon = \{t_1\}$$
$$(a_1.t_1) \,⪥|_r\, (a_2.t_2) = \{a_1.t \mid t \in t_1 \,⪥|_r\, (a_2.t_2)\}$$
$$\cup \left\{ a_2.t \,\middle|\, \begin{array}{l} t \in (a_1.t_1) \,⪥|_r\, t_2, \\ \neg(a_1.t_1 \,⪥|_r\, \theta(a_2)) \end{array} \right\}$$

This conditional sequencing operator entirely covers previous notions of weak sequencing and interleaving as we have $\parallel\, =\, ⪥|_L$ and $⪥\, =\, ⪥|_\emptyset$. Formal proofs of those statements are given in Appendix A.1 which follows the structure of the Coq proof available in (Mahe 2023a).

We denote by ; the concatenation (strict sequencing) operator on multi-traces, and by $\cup$ the set-theoretic union.

Then, in order to define the semantics of loops, we extend the Kleene star notation for our new scheduling operator. For any $\diamond \in \{;, \bar{\times}|_r\}$ and any set of traces T, the Kleene closure $T^{\diamond*}$ of T is defined by $T^{\diamond*} = \bigcup_{j\in\mathbb{N}} T^{\diamond j}$ with $T^{\diamond 0} = \{\varepsilon\}$ and $T^{\diamond j} = T \diamond T^{\diamond(j-1)}$ for $j > 0$.

Finally, we define a denotational semantics $\rho : \mathbb{I} \to \mathcal{P}(\mathbb{T}_{|L})$ of interactions as a set of global traces - i.e. traces defined on the partition $C_t = \{L\}$ - associated to a specific interaction term:

– $\rho(\varnothing) = \{\varepsilon\}$ and $\rho(a) = \{a\}$ for any action $a \in \mathbb{A}$
– and for any i_1 and i_2 in $\mathbb{I}$:

  - $\rho(alt(i_1, i_2) = \rho(i_1) \cup \rho(i_2)$
  - $\rho(strict(i_1, i_2)) = \rho(i_1); \rho(i_2)$
  - $\rho(coreg_r(i_1, i_2)) = \rho(i_1)\bar{\times}|_r\rho(i_2)$
  - $\rho(loop_S(i_1)) = \rho(i_1)^{;*}$
  - $\rho(loop_{C_r}(i_1)) = \rho(i_1)^{\bar{\times}|_r*}$

The pruning and execution relations are then characterized w.r.t. this denotational formulation in Th.1 and Th.2.

Theorem 1. *For any $L' \subseteq L$ and any i and i' from $\mathbb{I}$ we have:*

$$(i \xrightarrow{L'}_{\times} i') \quad \Rightarrow \quad (\rho(i') = \{t \in \rho(i) \mid \forall l \in L', \ \neg(t \bar{\times}|_\varnothing l)\})$$
$$(i \xrightarrow{L'}_{\times}) \quad \Rightarrow \quad (\forall t \in \rho(i), \exists l \in L', t \bar{\times}|_\varnothing l)$$

Proof. A detailed proof is given in Appendix A.2 and corresponds to the Coq proof available in (Mahe 2023a). □

Th.1 states that transformations $i \xrightarrow{L'}_{\times} i'$ characterize interactions i' which specify behaviors that are exactly those specified by i with no action occurring on a lifeline of L'. By contrast, $i \xrightarrow{L'}_{\times}$ stands for "all the behaviors of i involve at least one action occurring on a lifeline of L'". Let us observe that if we choose $L' = L$ in Def.4, the expression $i \xrightarrow{L}_{\times} \varnothing$ (resp. $i \xrightarrow{L}_{\times}$) means that the interaction i accepts (resp. does not accept) the empty multi-trace ε_C. We take advantage of this observation in Def.6 to provide a more compact presentation of the operational semantics than the one in (Mahe et al. 2022).

Theorem 2. *For any $i \in \mathbb{I}$, $a \in \mathbb{A}$ and $t \in \mathbb{T}_{|L}$:*

$$(a.t \in \rho(i)) \Leftrightarrow \left(\exists i' \in \mathbb{I}, \ (i \xrightarrow{a} i') \wedge (t \in \rho(i')) \right)$$

Proof. A detailed proof is given in Appendix A.3 and corresponds to the Coq proof available in (Mahe 2023a). □

Th.2 characterizes the execution relation $\to$ w.r.t. the denotational-style semantics ρ. It states that the follow ups i' s.t. $i \xrightarrow{a@p} i'$ indeed specify all the continuations of the behaviors specified by i.

Finally, in Th.3 we justify the correctness of our operational-style semantics from Def.6 for the particular case of the partition $C_t = \{L\}$ w.r.t. the denotational semantics ρ inspired by (Knapp & Mossakowski 2017).

Theorem 3. *For any $i \in \mathbb{I}$, we have $\sigma_{C_t}(i) = \rho(i)$*

Proof. Implied by Th.1 (for $L' = L$) and by Th.2. See Appendix A.4 and the Coq proof in (Mahe 2023a). □

Th. 3 states that both definitions of ρ and σ_{C_t} coincide on the trivial partition $C_t = \{L\}$ which preserves the most information on partial orders between events. For other partitions $C \in \text{Part}(L)$, the analogy of denotational and semantic semantics is obtained by observing that $\sigma_C(i)$ corresponds to a projection of $\sigma_{C_t}(i)$ from finer to coarser multi-traces, and thus corresponds also to the same projection applied on $\rho(i)$.

3.4. Application to multi-trace analysis

Accepted multi-traces of a certain interaction i are defined via their semantics $\sigma_C(i)$ in Def.6. If, in a practical setting, a multi-trace μ is observed during a system execution, the conformance of μ to i can be brought back to a problem of membership as in (Mahe et al. 2021) i.e. verifying whether or not $\mu \in \sigma_C(i)$.

In (Mahe et al. 2020), we have proposed an algorithm for analyzing traces w.r.t. interactions, which corresponds to a single co-localization in our present framework, i.e. with the trivial partition $C_t = \{L\}$. These analyses determine whether or not a behavior given as a trace is accepted. The principle of the algorithm is to consider the first element a_1 of the trace t to be analyzed (therefore of the form $a_1.t'$), to execute it in the reference interaction i_0 and to remove it from the trace. This allows us to start again with all the interactions i_1 verifying $i_0 \xrightarrow{a_1@p} i_1$ for (possibly several) positions p and the remaining trace t'. If the original trace t of length n can be emptied (via n such steps), then it means that it is accepted by the original interaction i_0 iff ε is accepted by the last interaction i_n. If this is not possible then this means that the behavior t deviates from i_0.

This principle can be directly adapted to multi-traces as demonstrated with the algorithm defined for the discrete partition $C_d = \{\{l\} \mid l \in L\}$ in (Mahe et al. 2021). The latter algorithm can be easily extended to consider any partition $C \in \text{Part}(L)$ and to consider global prefixes. Let us illustrate this with our running example i.e. let us analyze the multi-trace from Fig.1c w.r.t. the interaction from Fig.2 which semantics we have illustrated in Fig.4.

The application of the algorithm is represented on Fig.5. In order to analyze the multi-trace, we try to reconstruct a global behavior (global trace) from the execution tree (e.g. Fig.4) of the interaction which can be projected into the multi-trace. To that end, we use the execution relation $\to$ from Def.5 (the operational semantics). If an action is at the beginning of one of the multi-trace's local traces, and if it is immediately executable in the interaction model, the algorithm performs a step in which it both consumes
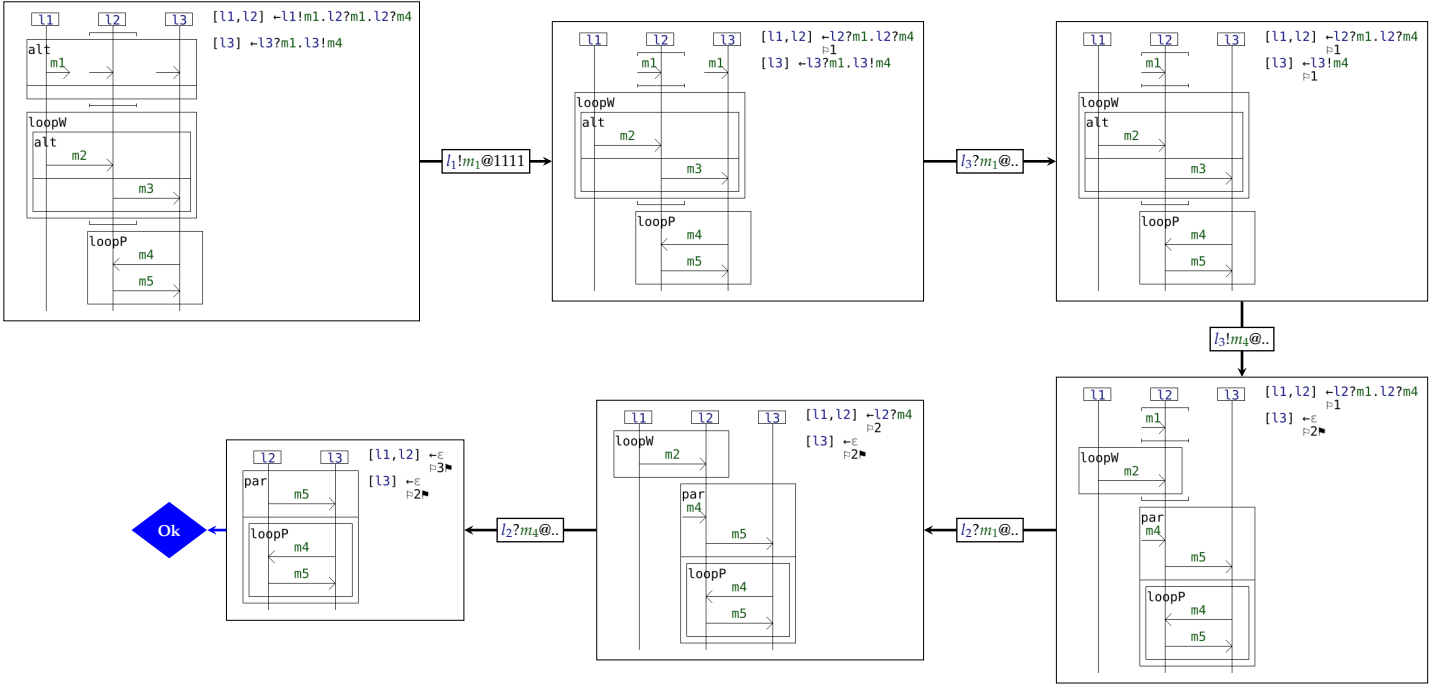
Figure 5 Illustrating multi-trace analysis algorithm adapted (to co-localizations) from (Mahe et al. 2021).

it from the multi-trace and executes it in the interaction. This enables one to replay a behavior characterized by the multi-trace in the model. If it is possible then it means that the multi-trace satisfies the specifying interaction. Otherwise this means that the multi-trace violates it. The analysis itself is represented graphically under the form of a graph on Fig.5. Every node on this graph contains both an interaction (on the left) and a multi-trace (on the right), the initial node containing the initial interaction i_0 (Fig.2) and multi-trace μ_0 (Fig.1c). As a visual aid, a ⌷ (white flag) symbol appears under components $c \in C$ of the multi-trace whenever, at this moment in the reproduction of the behavior in the model, observation has started on c. The number following the flag then represents the number of actions which have been observed. A ⌷ (black flag) symbol marks the end of observation.

However, this general principle is insufficient for analyzing multi-traces in the case of partial observability. In this context, *partial observation* signifies that the multi-trace logged by the instrumentation does not characterize the entire execution of the DS. More concretely, some events may be missing from the multi-trace w.r.t. an ideal multi-trace which would have been observed with ideal conditions of observation. The notion of *multi-trace slice* from Sec.2.2 proposes a specific definition of partial observation, where events may be missing at the beginning and/or the end of each local trace component of the multi-trace (independently).

As a means to understand how partial observation is challenging for RV we consider the example from Fig.1d, which is a partial observation of the multi-trace from Fig.1c. If we were to reorder actions globally, this observation

$$l_1!m_1.l_3?m_1.l_2?m_1.l_3!m_4.l_2?m_4$$

(a) Missing actions in the multi-trace from Fig.1d transposed in the global scenario from Fig.1a
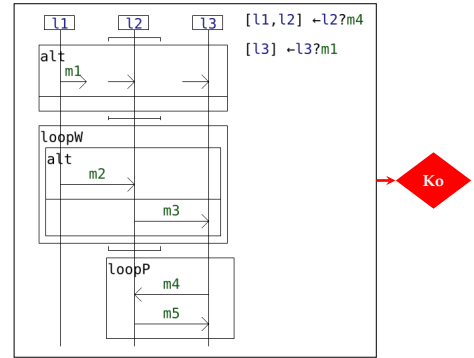
(b) The analysis yields a *Fail* because of unobserved actions

Figure 6 Limitation of the approach from (Mahe et al. 2021) under partial observation.

could be described as in Fig.6a. Missing actions, i.e. actions that are not observed by the instrumentation, are not necessarily at the beginning or the end globally but there may be several sub-words missing from the global trace (here $l_1!m_1$ and $l_2?m_1.l_3!m_4$ inserted in light gray in the global trace given in Fig.6a). The fact that such missing actions may be located anywhere in a globally sequential behavior (here the broadcast of m_1 from l_1 to l_2 and l_3 followed by the passing of m_4 from l_3 to l_2) is particularly challenging for ORV. Using the algorithm from (Mahe et al. 2021) this would yield a *Fail* verdict - as illustrated

on Fig.6b - because this algorithm cannot differentiate between the system going out of specification and it being partially observed.

This motivates the definition of an ORV algorithm that is tolerant to partial observation, which is the object of the next section.

4. ORV algorithm with bounded simulation

As we have seen with the example from Fig.1d and Fig.6, DS executions can be partially observed due to issues of synchronization between local observers. As a result, a correct behavior may be observed as a slice $\mu' \in \widetilde{\mu}$ of an accepted multi-trace $\mu \in \sigma_C(i)$ with missing elements at the beginning and/or the end of the traces corresponding to co-localizations of C. Then, because we might have $\mu' \notin \sigma_C(i)$, membership is not enough to verify conformance. The property which we have to verify is rather whether or not $\mu' \in \widetilde{\sigma_C(i)}$.

Simulation is a straightforward answer to partial observation in so far as actions missing from the multi-trace may simply be simulated in the model. In order to identify a slice μ' of an accepted multi-trace, we may simulate actions a that occur either before the first action of the corresponding component $\mu'_{|\theta_C(a)}$ or after its last action i.e. outside of the period of (continuous) observation of the component. Simulating such actions hopefully enables the consumption of further actions in the multi-trace (in the same component or any other). The approach that we propose is optimistic in so far as that it suffices that there exists some missing actions that can be be simulated which explain the observed behavior. In other words, this means that if, during an execution, what we observe of it can be explained by a behavior without violations of the specification, then, it is accepted even though a violation might have happened in some unoberseved part of the behavior.

Simulation explores possible missing actions that could have been executed in order to explain the behavior observed via the multi-trace w.r.t. the specifying interaction. However, the presence of loops in interaction models makes it possible to simulate arbitrarily many actions, making a naive simulation-based algorithm non-terminating. As a practical solution, we should bind simulation up to a certain criterion so that we can hope to find missing actions within a finite search space.

Defining pertinent stopping criteria on simulation being no trivial matter, we firstly formalize a simulation-based algorithm using an abstract criterion.

4.1. Algorithm initialization

Our algorithm relies on two mechanisms: one for executing actions and consuming them from the multi-trace and one for simulating actions without a corresponding consumption. It then consists in exploring a graph to find possible explanations of an observed multi-trace. From a finite number of starting nodes, the mechanism of execution can only yield a finite number of steps given that the multi-trace is finite (in terms of number of actions) and, for each such action, there can only be finitely many manners to interpret it in the interaction term (see (Mahe et al. 2021)). However, this is not the case for the mechanism of simulation. In order for the algorithm to be terminating, we have to limit the simulation steps using a criterion.

We define a generic algorithm which relies on three notions:

- flags defined by a function γ which aim is to keep track of whether or not observation has started on the co-localizations $c \in C$;
- a space of measures $\mathbb{J}$ fitted with a strict order relation $<_{\mathbb{J}}$ which can be parameterized and serves as a means to limit the number of simulation steps;
- an initialization function κ which sets the initial value of the measure at the start of the algorithm process.

Flags γ In the algorithm, execution steps and simulation steps may be interleaved. Execution steps can be taken at any moment, provided that there is a match between an action at the beginning of the multi-trace and an immediately executable action of the interaction. However, an additional condition is required for applying a simulation step. Indeed, the goal of simulation is to reconstruct parts of behaviors that were not observed at the beginning and the end of the period of observation on a given co-localization. As a result, we need an additional condition to ascertain that we are outside this period of observation.

To that end, we define $\gamma : C \to \mathbb{B}$ (where $\mathbb{B} = \{\bot, \top\}$ is the usual set of Boolean values) such that for any $c \in C$, $\gamma(c) = \bot$ if observation has not started and $\gamma(c) = \top$ if it has. We denote by $\gamma_\bot$ the case where observation has not started on any co-localization i.e. $\forall c \in C, \gamma_\bot(c) = \bot$. For any $\gamma \in \mathbb{B}^C$ and $c \in C$, we denote by $\gamma + c$ the function s.t. $\forall c' \in C \setminus \{c\}, (\gamma + c)(c') = \gamma(c')$ and $(\gamma + c)(c) = \top$. We use this notation to update the observation status on c (i.e. that it has started).

Parameterizable measure $\mathbb{J}$ In the sequel, we consider:

- a set $\mathbb{J}$ of measures fitted with a strict order relation $<_{\mathbb{J}}$ which admits no infinite descending chains, i.e. which admits no infinite sequences $(j_i)_{\in \mathbb{N}}$ of elements in $\mathbb{J}$ verifying $\forall i \in \mathbb{N}, j_{i+1} < j_i$;
- and a relation $\to \subseteq \mathbb{J} \times (\mathbb{I} \times \{1, 2\}^*) \times \mathbb{J}$ verifying that for any $j, j' \in \mathbb{J}$, $i \in \mathbb{I}$ and $p \in \{1, 2\}^*$, if $j \xrightarrow{i,p} j'$ then $j' <_{\mathbb{J}} j$. We note $j \xrightarrow{i,p} \not{}$ if there does not exist j' s.t. $j \xrightarrow{i,p} j'$.

From a pair (i, j) where i is an interaction and j a measure, given an action at position p in i, if $j \xrightarrow{i,p} \not{}$ then we cannot simulate this action starting form i. If however there exists j' s.t. $j \xrightarrow{i,p} j'$ then we may simulate it and,

given $i \xrightarrow{a@p} i'$ with a action at position p in i, reach the pair (i', j'). In this manner, we can bind simulation to a strictly decreasing measure. In the context of our ORV algorithm, and because $(\mathbb{J}, <_{\mathbb{J}})$ has no infinite descending chain, this will imply that, in any run of our algorithm, there can only be finitely many consecutive steps of simulation.

Example 1. *For instance, we can consider $\mathbb{J} = \mathbb{N}$ (positive integers), $<$ the classical inequality on positive integers, and the relation $\rightarrow$ s.t. for any $j \in \mathbb{N}$, any $i \in \mathbb{I}$ and p s.t. $\exists i' \in \mathbb{I}$ s.t. $i \xrightarrow{i|p} i'$ we have $j \xrightarrow{i,p} j - 1$. With this example, we decrement by 1 each time we simulate any action.*

Parameterizable measure (re-)initialization κ The measure being now defined, it still needs to be initialized. For that, we consider any arbitrary function $\kappa : \mathbb{I} \rightarrow \mathbb{J}$. Because consecutive sequences of unobserved actions can occur in between observed actions, whenever an action is executed and consumed in the multi-trace instead of being simulated, we may reset the measure to $\kappa(i')$ given the execution $i \xrightarrow{a@p} i'$.

Example 2. *For instance, following Ex.1, we can consider κ s.t. for any $i \in \mathbb{I}$ we have $\kappa(i) = 5$. With this example criterion, we may simulate successively a number of actions which is at most 5.*

In practice, however, so as to be able to define a wide variety of criteria, the definition of κ may also depend on other variables (besides the interaction i) such as the (size of the) multi-trace μ which is analyzed, the current state of the flags γ etc. This is reflected in the tool implementation. However, for the sake of simplicity, we consider $\mathbb{I} \rightarrow \mathbb{J}$ as the signature of κ.

4.2. Analysis graph

Let us define a directed search graph $\mathbb{G}$ which set of vertices is $\mathbb{V} = (\mathbb{I} \times \mathbb{M}_C \times \mathbb{B}^C \times \mathbb{J}) \cup \{Ok, Ko\}$ i.e. which vertices are:

– either of the form (i, μ, γ, j) with i an interaction, μ a multi-trace, γ a flag and j a measure (the latter two defined as in Section 4.1);
– or can be one of the two verdicts Ok or Ko.

The arcs of $\mathbb{G}$ are defined using 4 rules: R_p (for "pass verdict"), R_f (for "fail verdict"), R_e (for "execute") and R_s (for "simulate") which are defined in Def.7.

Definition 7. *The graph $\mathbb{G} = (\mathbb{V}, \leadsto)$ is defined by:*
– the set $\mathbb{V} = (\mathbb{I} \times \mathbb{M}_C \times \mathbb{B}^C \times \mathbb{J}) \cup \{Ok, Ko\}$
– the relation $\leadsto \subseteq \mathbb{V} \times \mathbb{V}$ s.t. $\forall v, v' \in \mathbb{V}, v \leadsto v'$ holds iff it can be derived by applying[2] the four following rules:

$$(R_p) \; \frac{}{(i, \varepsilon_C, \gamma, j) \leadsto Ok}$$

---
[2] the notation $(R) \frac{H}{v \leadsto v'}$ signifies that $v \leadsto v'$ can be inferred by applying rule R if we suppose that hypothesis H holds.

$$(R_e) \; \frac{\exists\, p \text{ s.t. } i \xrightarrow{a@p} i'}{(i, a \vec{\odot} \mu, \gamma, j) \leadsto (i', \mu, \gamma + \theta_C(a), \kappa(i'))}$$

$$(R_s) \; \frac{(\mu \neq \varepsilon_C) \wedge \left(\exists\, a, p \text{ s.t.} \left| \begin{array}{l} (i \xrightarrow{a@p} i') \\ \wedge (j \xrightarrow{i,p} j') \\ \wedge \left(\begin{array}{l} (\gamma(\theta_C(a)) = \bot) \\ \vee (\mu_{|\theta_C(a)} = \varepsilon) \end{array} \right) \end{array} \right. \right)}{(i, \mu, \gamma, j) \leadsto (i', \mu, \gamma, j')}$$

$$(R_f) \; \frac{(\mu \neq \varepsilon_C) \wedge \left(\forall\, a, p, i' \text{ s.t.} \left| \begin{array}{l} (\nexists\, \mu' \text{ s.t. } \mu = a \vec{\odot} \mu') \\ \wedge \left(\begin{array}{l} (j \xrightarrow{i,p}) \\ \vee \left(\begin{array}{l} (\gamma(\theta_C(a)) = \top) \\ \wedge (\mu_{|\theta_C(a)} \neq \varepsilon) \end{array} \right) \end{array} \right) \end{array} \right. \right)}{(i, \mu, \gamma, j) \leadsto Ko}$$

where i and i' are interactions, μ and μ' are multi-traces, γ is a flag, j and j' are measures, a is an action and p is a position.

Rules R_p and R_f define edges from nodes of the form (i, μ, γ, j) to verdicts (i.e. resp. Ok and Ko). Their conditions of application are exclusive to that of all the other rules which imply that if $v \leadsto Ok$ (or $v \leadsto Ko$) then there is no other edge originating from v.

– Given that the objective of the algorithm is to recognize slices of accepted multi-traces and because the empty multi-trace ε_C is a slice of any other multi-trace, rule R_p (for "pass") yields a Ok verdict.
– R_f (for "fail") yields a Ko verdict if the multi-trace is not empty and if it is not possible to apply any of the other two rules R_e or R_s.

Based on the machinery of execution $i \xrightarrow{a@p} i'$ (Def.5), rules R_e and R_s specify edges of $\mathbb{G}$ of the form $(i, \mu, \gamma, j) \leadsto (i', \mu', \gamma', j')$ in which an action occurs:

– the application of R_e corresponds to the identification of an action a which can simultaneously be consumed at the head of a component of μ and be executed from i. When rule R_e applies, γ is updated into $\gamma + \theta_C(a)$ to reflect that observation has started on the co-localization $\theta_C(a)$ on which action a has been observed.
– with R_s the action is simulated in the interaction without a corresponding consumption in the multi-trace. Action a can be simulated if and only if at this moment in the global scenario we are outside of the period of observation on the corresponding co-localization $\theta_C(a)$ i.e. it has either not started ($\gamma(\theta_C(a)) = \bot$) or has already finished ($\mu_{|\theta_C(a)} = \varepsilon$).

Note that the conditions of application of R_e and R_s are not mutually exclusive. The same action a may be either executed or simulated in case observation has not yet started on $\theta_C(a)$. This allows considering any missing prefix of the multi-trace.

Theorem 4 (Finite reachable sub-graph). *From any vertex* $v = (i, \mu, \gamma, j)$, *the sub-graph of* $\mathbb{G}$ *reachable from* v *is finite.*

Proof. We prove this in two steps: **(1)** all paths in $\mathbb{G}$ from v are of finite length and **(2)** there is a finite number of distinct paths in $\mathbb{G}$ from v.

To prove **(1)** this let us consider a measure on vertices of $\mathbb{V}$ given as a tuple by $|Ok| = |Ko| = (-1, j_0)$ (with j_0 any element of $\mathbb{J}$) and, for any vertex v of the form (i, μ, γ, j), $|v| = (|\mu|, j)$ where $|\mu|$ is the length of the multi-trace (in total number of actions). Considering the lexicographic order on the tuple (with the relations $<$ on integers for the first element and $<_\mathbb{J}$ for the second), each one of the 4 rules decreases strictly this measure:

- for R_p and R_f because $(-1, j_0) < (|\mu|, j)$ given that $|\mu| \geq 0$ for any multi-trace
- for R_e because $(|\mu|, \kappa(i')) < (|a \; \vec{\odot} \; \mu|, j)$ given that $|a \; \vec{\odot} \; \mu| = |\mu| + 1$
- for R_s because $(|\mu|, j') < (|\mu|, j)$ given that $j \xrightarrow{i,p} j'$ by the definition of the rule R_s.

In addition $(-1, j_0)$ is a global minimum for this measure and because there are no infinite descending chains in $\mathbb{J}$, there are also none for this measure on nodes of $\mathbb{G}$. Hence, by construction any path in $\mathbb{G}$ is finite.

To prove **(2)** we remark that, from any given node $v = (i, \mu, \gamma, j)$ there are at most $2 * |i| + 1$ (where $|i|$ designates the total number of actions in i) edges of the form $v \rightsquigarrow v'$ with $v' \notin \{Ok, Ko\}$ (this is in the worst case, when every action can be both executed and simulated). $\qquad\square$

Th.4 states that only a finite sub-graph of $\mathbb{G}$ can be reached from any given vertex $v \in \mathbb{G}$. Let us also remark that any sink (i.e. any vertex without any outgoing transition) of $\mathbb{G}$ must either be Ok or Ko. They are indeed sinks because there are no rules specifying edges of the form $Ok \rightsquigarrow v$ or $Ko \rightsquigarrow v$ and they are the only ones because for any vertex of the form $v = (i, \mu, \gamma, j)$, if $\mu = \varepsilon_C$ then R_p applies and v is not a sink and if $\mu \neq \varepsilon_C$ then: **(1)** if there is a match between an action that can be executed from i and the head of a component of the multi-trace then rule R_e applies. **(2)** if there is some action of i that can be simulated, then rule R_s applies. **(3)** if neither condition 1 nor condition 2 hold then rule R_f applies. Indeed, by construction, the conditions of application of R_f are defined as complementary to the conditions of application of the other 3 rules R_p, R_e and R_s.

4.3. Verdicts and properties of the generic algorithm

The ORV algorithm consists in exploring reachable vertices of a graph $\mathbb{G}$ using $\rightsquigarrow$ (cf. Def.7) from an initial vertex $v = (i, \mu, \gamma_\perp, \kappa(i))$ where μ is the multi-trace which we want to analyze, i is the interaction which serves as the reference specification and $\gamma_\perp$ is the flag set to $\perp$ on each co-localization $c \in C$. In this starting node, we choose the flag $\gamma_\perp$ since observation has not started on any component

and we initialize the measure for simulation using any arbitrary function $\kappa : \mathbb{I} \to \mathbb{J}$.

This algorithm is generic given that the measures $\mathbb{J}$, the relation $\to$ and the functions κ are kept generic i.e. are only defined through their profiles and properties.

Because of Th.4 and because Ok and Ko are the only two possible sinks, we can conclude that at least one of them is reachable from v. In any case, because the reachable part of $\mathbb{G}$ is finite, it is always possible to determine in finite time if Ok is reachable from v (which we may denote by $v \overset{*}{\rightsquigarrow} Ok$) or not.

In the context of our ORV algorithm we return a *Pass* if we can ascertain that $v \overset{*}{\rightsquigarrow} Ok$ and an *Inconc* (meaning an inconclusive verdict) otherwise. The algorithm is defined as a function ω in Def.8 and is well-founded given the previous remark linked to Th.4.

Definition 8. *For any $C \in Part(L)$, we define $\omega_C : \mathbb{I} \times \mathbb{M}_C \to \{Pass, Inconc\}$ s.t. for any $i \in \mathbb{I}$ and $\mu \in \mathbb{M}_C$:*

- *$\omega_C(i, \mu) = Pass$ iff there exists a path in $\mathbb{G}$ from $(i, \mu, \gamma_\perp, \kappa(i))$ to Ok*
- *$\omega_C(i, \mu) = Inconc$ otherwise*

Th.5 states that a *Pass* verdict ensure the identification of a multi-trace as a slice of an accepted multi-trace of the initial interaction. However this property is not as strong as e.g. the correctness of the algorithm from (Mahe et al. 2021) i.e. we are not guaranteed to have a *Pass* for any and all slices of an accepted multi-trace.

Theorem 5 (Soundness). *For any interaction $i \in \mathbb{I}$, partition $C \in Part(L)$ and multi-trace $\mu \in \mathbb{M}_C$, we have:*

$$(\mu \in \sigma_C(i)) \Rightarrow (\omega_C(i, \mu) = Pass)$$
$$(\omega_C(i, \mu) = Pass) \Rightarrow (\mu \in \widetilde{\sigma_C(i)})$$

Proof. For the first point: if the multi-trace μ that is analyzed is in $\sigma_C(i)$ then there is a corresponding path in the execution tree of i (via Def 6, see also Fig.4). Using only rule R_e (and never R_s) it is then possible to consume μ in its entirety. As the application of R_e is not constrained by the value of the measure j, the analysis is close to that of the algorithm from (Mahe et al. 2021).

For the second point we can reason as follows. Given the nature of R_e and R_s, which both execute actions in the current interaction, any path $v = (i, \mu, \gamma_\perp, \kappa(i)) \overset{*}{\rightsquigarrow} Ok$ exactly corresponds to a path in the execution tree (see Fig.4) of i and hence to a full multi-trace μ_0 accepted by i i.e. s.t. $\mu_0 \in \sigma_C(i)$. Then, μ is a partial observation (i.e. a slice) of μ_0 given that: **(1)** it contains all the actions of μ corresponding to R_e steps while those corresponding to R_s are missing and **(2)** R_s being only applicable outside the period of observation of the components of μ, those missing actions are either before the start or after the end of those corresponding components. Hence $\mu \in \widetilde{\mu_0}$ and therefore $\mu \in \widetilde{\sigma_C(i)}$. $\qquad\square$

While the *Pass* verdict ensures that the multi-trace analyzed is indeed a slice of the considered interaction, the other global verdict *Inconc* only means an inconclusive verdict and not a *Fail* one because it does not necessarily mean that the multi-trace which is analyzed is not a slice of an accepted multi-trace. Depending on the structure of the input interaction and depending on the criterion that is selected to bind the simulation, we may not have simulated enough the interaction to bring it to states which would allow the entire consumption of the multi-trace.

5. Implementation and assessment

5.1. Instantiating the parameters

Our approach for simulation-based analysis, as presented in Sec.4, requires a parameterization to be concretized. It consists in defining $\mathbb{J}$, $\to$ and κ (in Sec.4 those were only characterized through their properties).

A variety of criteria could be used. For instance, with $\mathbb{J} = \mathbb{N}$ (positive integers), we could use an arbitrary maximum number of actions that can be successively simulated. And whenever an action $i_{|p}$, at position p in interaction i is executed, we would have $j \xrightarrow{i,p} j-1$.

Many such trivial criteria may be defined. However, in the following, we propose a slightly more subtle criteria which tries to strike a balance between a good coverage rate (the ability to identify most if not all slices) and efficiency (via taking care of not simulating too many actions, and hence decreasing the complexity/size of graph $\mathbb{G}$).

Let us consider a concrete criterion for binding simulation in the form of a tuple of integers (i.e. $\mathbb{J} = \mathbb{N}^2$) which we denote by (λ, α) such that:

- λ represents a maximum number of loops which can be instantiated in a consecutive sequence of simulation steps
- and α relates to a number of actions.

This set $\mathbb{J}$ is fitted with the lexicographic order.

In order to initialize and update this criterion, let us consider two functions: $\eta : \mathbb{I} \to \mathbb{N}$ which gives the total number of actions occurring outside loops and $\beta : \mathbb{I} \to \mathbb{N}$ which gives the maximum depth of nested loops. More precisely, $\beta(i) = max_{p \in pos(i)} \ \beta(i,p)$ where $\beta(i,p)$ is the number of nested loops above position p in interaction i (see Appendix B for a complete definition). We then define $\kappa(i)$ as the couple $(\beta(i), \eta(i))$.

We define the relation $\to \subseteq \mathbb{J} \times (\mathbb{I} \times \{1,2\}^*) \times \mathbb{J}$ by: for any $(\lambda, \alpha) \in \mathbb{J}$, any $i, i' \in \mathbb{I}$, $a \in \mathbb{A}$ and $p \in pos(i)$ s.t. $i \xrightarrow{a@p} i'$,

- if $\beta(i,p) = 0$, we are in the case where we simulate an action outside a loop. Here λ stays the same while α decreases. Indeed at least one action (the one which is executed) is removed from i (more may be removed due to pruning or choosing alternatives etc) and none are added (because no loops are instantiated). Here we have $(\lambda, \alpha) \xrightarrow{i,p} (\lambda, \eta(i'))$ with $0 \leq \eta(i') \leq \alpha - 1$ by construction.

- if $\beta(i,p) > 0$, we instantiate $\beta(i,p)$ loops which requires that $\lambda \geq \beta(i,p)$. Then $(\lambda, \alpha) \xrightarrow{i,p} (\lambda - \beta(i,p), \eta(i'))$. Here we reset the value of α because loop instantiation may change the total number of actions outside loops.

λ guarantees that we can instantiate at least once every loop in the interaction (although not necessarily in the same path). This limit on the number of loops is sufficient to guarantee termination because there can only be a finite number of actions existing outside loops and each of those can be simulated at least once (which corresponds to α). Our definition of $\mathbb{J}$ and $\to$ ensures the required properties, i.e. that it is strictly decreasing within a space which has no infinite descending chains. This guarantees the termination of the parameterized algorithm.

This proposal for κ is independent of the size of the analyzed multi-trace. It only depends on the reference interaction. This is advantageous in so far as, in practice, the size of the interaction is small compared to the size of the multi-trace. Thus, this measure allows one to calibrate the number of simulation steps according to the complexity of the interaction while drastically limiting their number.

Once κ is properly defined, the multi-trace analysis algorithm essentially boils down to a traversal of a finite graph. Different traversal heuristics (depth/breadth-first, with priorities on the application of the rules etc.) can be implemented. This is mentioned in Sec.7.3 in the context of our tool HIBOU.

In the following, we illustrate on a small example how κ comes into play for the construction of the graph $\mathbb{G}$.

5.2. Illustration on the running example

Applying our algorithm with this criterion on our initial example i.e. analyzing the multi-trace μ from Fig.1d against the interaction i from Fig.2 yields the graph (partially drawn) on Fig.7. In this particular case, we effectively conclude that μ is a slice of a behavior accepted by i. Here, the global scenario which is reconstructed during the analysis and that match μ corresponds to the path leading to Ok displayed on Fig.7. This path corresponds to the trace $l_1!m_1.l_3?m_1.l_3!m_4.l_2?m_1.l_2?m_4$. Notice that is uses a different interleaving of the simulated actions $l_3!m_4$ and $l_2?m_1$ w.r.t. the global scenario described on Fig.1a and Fig.6a.

Let us comment this path in the graph. Each vertex in this path is annotated with a circled number (from ① to ⑤). Vertex ① corresponds to the initial vertex $(i_0, \mu_0, \gamma_\perp, \kappa(i_0) = (\lambda_0, \alpha_0))$ from which the analysis starts, where μ_0 is the input multi-trace from Fig.1d which we want to analyze and i_0 is the input interaction from Fig.2 which serves as a specification. We initialize the measure to $\kappa(i_0) = (\beta(i_0), \eta(i_0))$, hence $\lambda_0 = \beta(i_0) = 1$ because the maximum depth of nested loops is 1 and $\alpha_0 = \eta(i_0) = 3$ because there are 3 actions outside loops.
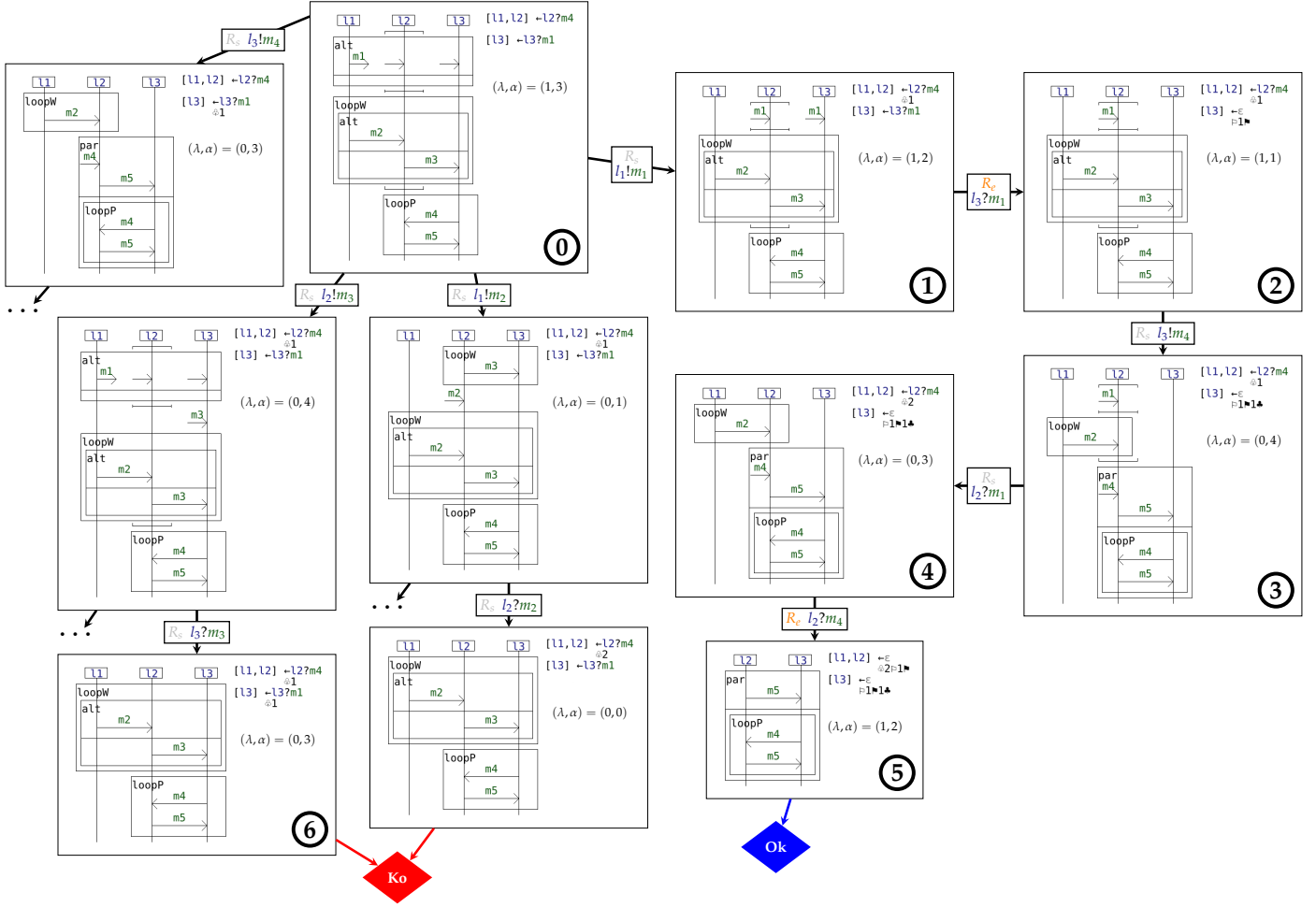
Figure 7 Multi-trace slice analysis

From the initial vertex ⓪, rules R_p and R_e cannot be applied. Indeed, R_p is not applicable because the multi-trace is not empty. As for R_e, it is because we cannot execute in the interaction any of the actions that are at the heads of the components of the multi-trace (i.e. neither $l_2?m_4$ nor $l_3?m_1$ can be immediately executed, cf. Fig.4 which enumerates all immediately executable actions). However, R_s can be applied on any of the four immediately executable actions (cf. Fig.4). Actions $l_3!m_4$, $l_2!m_3$ and $l_1!m_2$ are at depth 1 w.r.t. loops. Hence, because $\lambda_0 = 1$ we can simulate them. Action $l_1!m_1$ is outside all loops and can be simulated because $\alpha_0 = 3$. Because R_s can be applied, R_f cannot be applied from ⓪.

The simulation of action $l_1!m_1$ (at position 1111) leads to vertex ①. In this vertex $(i_1, \mu_1, \gamma_1, (\lambda_1, \alpha_1))$ we have i_1 s.t. $i_0 \xrightarrow{l_1!m_1@1111} i_1$ because the corresponding action is executed in the model, $\mu_1 = \mu_0$ because no action has been consumed in the multi-trace, $\gamma_1 = \gamma_0$ because observation has not started on any additional co-localization, $\lambda_1 = \lambda_0$ because no action inside a loop has been simulated and $\alpha_1 = \alpha_0 - 1 = 2$ because one action outside loops has been simulated. Here, we have one step of simulation before the start of observation on component $\{l_1, l_2\}$. As a visual aid,

on Fig.7 a ♧ (white clover) symbol indicates the beginning of simulation before the start of observation. This ♧ is then followed by the number of simulated actions.

From vertex ①, rule R_e can be applied on action $l_3?m_1$ i.e. we can execute this action and consume it from the multi-trace. The previous step of simulation has put the initial model i_0 to a state i_1 from which this action can be executed. In the next vertex ②, $\gamma_2(\{l_3\}) = \top$ because observation has then started on the co-localization $\{l_3\}$. Moreover, because the entire trace component on $\{l_3\}$ has been consumed, we have on Fig.7 both the ⊨ and ⊨ visual aids. In ② the measure is also reset to $\lambda_2 = 1$ and $\alpha_2 = 1$ because of the loop depth and number of actions outside loops in i_2.

From vertex ②, neither R_p nor R_e can be applied. However, it is possible to simulate actions $l_1!m_2$, $l_2?m_1$, $l_2!m_3$ and $l_3!m_4$ because of the current values of λ_2 and α_2 and for the following reasons: for the first three actions, simulation is possible because observation has not yet started on co-localization $\{l_1, l_2\}$; for action $l_3!m_4$, it is possible because observation has already ended on co-localization $\{l_3\}$ (because the corresponding trace component is already empty).

The simulation of $l_3!m_4$ leads to vertex ③. On Fig.7, the visual aid ♣ (black clover) denotes the number of simulation steps after the end of observation. The measure is updated so that $\lambda_3 = \lambda_2 - 1 = 0$ because $l_3!m_4$ was at depth 1 (hence it is not possible to instantiate new loops in simulation) and $\alpha_3 = 4$ because there are 4 actions outside loops in i_3.

From vertex ③ an additional simulation step leads to ④. Because the simulated action $l_2?m_1$ is outside all loops, it is α which is decremented. Finally, from vertex ④, rule R_e can be applied so that the entire multi-trace is emptied in vertex ⑤. Then, Ok can be reached by the application of R_p from ⑤.

Due to having several choices in the applications of rules R_e and R_s, several distinct paths may be opened during the analysis (as illustrated by the paths towards the left of Fig.7 which leads to Ko and by the $\cdots$ representing other paths which are not drawn). However, via the use of some heuristics and by terminating the analysis as soon as a Ok is reach, one can limit the size of the part of $\mathbb{G}$ which is explored.

At the bottom of Fig.7 we have also annotated one of the vertices as ⑥ so as to illustrate the application of rule R_f. Here, the two previous simulation steps have lead to a vertex $(i_6, \mu_6, \gamma_6, (\lambda_6, \alpha_6))$ in which: **(1)** the multi-trace is not empty i.e. $\mu_6 \neq \varepsilon_C$, **(2)** there are no immediately executable actions that match the heads of μ_6 and **(3)** we have $\lambda_6 = 0$ and there are no actions outside loops remaining in the interaction. Therefore, neither R_p, R_e nor R_s can be applied. As a result, we apply rule R_f which leads to Ko.

5.3. Further remarks on the approach

$$l_1!m_1.l_1!m_1.l_2?m_1.l_2?m_1.l_1!m_2.l_2?m_2$$

(b) Global execution which occurred

$$\begin{cases} [l_1] & l_1!m_1 \\ [l_2] & l_2?m_2 \end{cases}$$

(c) Observed multi-trace

$$l_1!m_1.l_1!m_2.l_2?m_1.l_2?m_2$$

(d) Minimal reconstructible global execution

(a) Specifying diagram

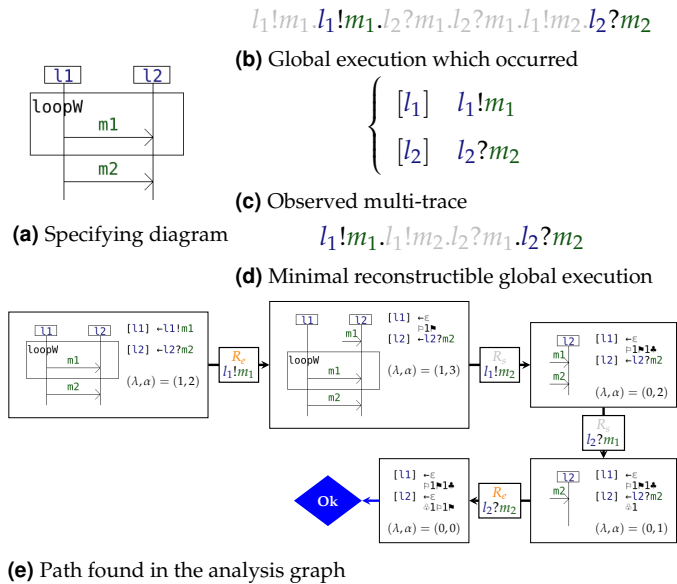**(e)** Path found in the analysis graph

Figure 8 Filling-in missing sections optimistically & a minima

Simulation steps are used to find possible replacements for missing sections in what is observed of the global scenario. As we have seen, depending on the architecture of the system, those missing sections can be temporally interspersed in-between sections that are observed in the global scenario.

Another example is described on Fig.8. Here a global execution characterized by the trace from Fig.8b was observed and recorded into the multi-trace from Fig.8c. The observation on l_1 both started too late and ended too early while that on l_2 started too late. The actions which have not been observed are grayed-out in Fig.8b. The global execution is correct w.r.t. the input interaction from Fig.8a and hence this multi-trace is an accepted slice. Analyzing it w.r.t. the interaction yields the graph given on Fig.8e. The algorithm tries to find a global scenario which fits the observed multi-trace. However, this global scenario may not necessarily correspond to the one which effectively occurred. It is sufficient that it both conforms to the specification and explains the observed multi-trace. Ideally it should then look for such a minimal reconstructible scenario such as the one from Fig.8d which indeed corresponds to the one unveiled by the analysis graph on Fig.8e.

We may remark that our simulation-based approach is optimistic. Indeed, it suffices to find a reconstructible global scenario that both fits the interaction model and the input multi-trace. It may be possible that unwanted behaviors occurred but their detection is not possible from what was observed of the execution. From another perspective, the simulation is optimistic because we only simulate actions which do not deviate from the specification.

The conditions for the application of rules R_e and R_s are not mutually exclusive. Hence the same action at the same position within an interaction might be both simulated or executed from the same vertex. We consider an example global scenario in which a first message m_1 is transmitted from l_1 to l_2 and then a second one transits from l_1 to l_2 before a final message m_2 is send from l_2 to l_1. We then suppose that this behavior is observed through the multi-trace which is analyzed on Fig.9. In this multi-trace, only the second instance of $l_1!m_1$ is observed. If we start the analysis by executing $l_1!m_1$, we consume the second instance of $l_1!m_1$ from the multi-trace instead of the first. This later leads to a Ko as illustrated by the path on the left of Fig.9. The correct first step in Fig.9 is to simulate $l_1!m_1$ instead of executing it. It is therefore important to allow the same action to be both simulated and executed, which explains having non mutually-exclusive conditions for the application of rules R_e and R_s.

We may also remark that the problem of speculating which actions to simulate incurs a high complexity. The solution which we have presented here tackles this problem using brute force because we explore possible simulated actions exhaustively up to a certain bound which makes the search space finite. Various optimizations can be envisioned e.g. via a static analysis of the multi-trace when choosing between alternative branches, loops, etc.

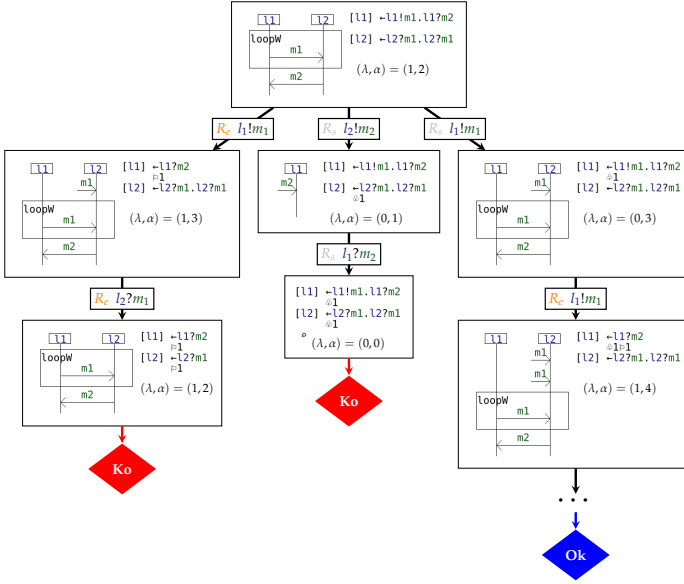In the algorithm, transitions allowed by rule R_e are of

Figure 9 Pertinence of non mutually exclusive R_e and R_s

(a) Simulating enough

(b) Not simulating too much

Figure 10 Motivation behind the reset of the measure

the form $(i, a \vec{\odot} \mu, \gamma, j) \rightsquigarrow (i', \mu, \gamma + \theta_C(a), \kappa(i'))$ i.e. we reset the measure j to a new value which depends on the follow-up interaction i'. With the examples on Fig.10, we illustrate the motivation behind this reset of the measure.

In Fig.10a, we simulate two instances of $l_1!m_1$ in order to be able to execute the two instances of $l_2?m_1$ in the multi-trace. Instead of using an arbitrary criterion of size 2 for the number of loops, we can use our proposal criterion on the maximum loop depth (which is 1 here) but reset it every time an action is executed. Here, because lifelines l_1 and l_2 are on different co-localizations, this allows alternating between steps of simulation on co-localization $\{l_1\}$ and steps of execution on co-localization $\{l_2\}$ so as to consume the multi-trace in its entirety. Here, the reset of the measure enables us to recognize any repetition of $l_2?m_1$. With this reset, the initialization of the measure does not need to depend on the size of the multi-trace μ but only on the structure of the interaction i.

In the analysis from Fig.10b, the initial interaction has a maximum loop depth of 2. However, due to pruning operations, after the first step, which is an execution (rule R_e), the follow-up interaction only has a maximum loop depth of 1. Hence it is pertinent to reset the measure to reflect this change and so as not to allow more simulation than necessary.

5.4. Limitations related to inconclusiveness

Our algorithm returns either a *Pass* verdict or an *Inconc* verdict. When a *Pass* verdict is returned, the analyzed multi-trace is indeed a slice of an accepted multi-trace. However, the *Inconc* verdict do not necessarily reflect a failure. This is because, by bounding the number of simulation steps, it may well be that a correct slice can not be recognized, as its recognition would have required more simulation steps. In particular, the use of certain specific
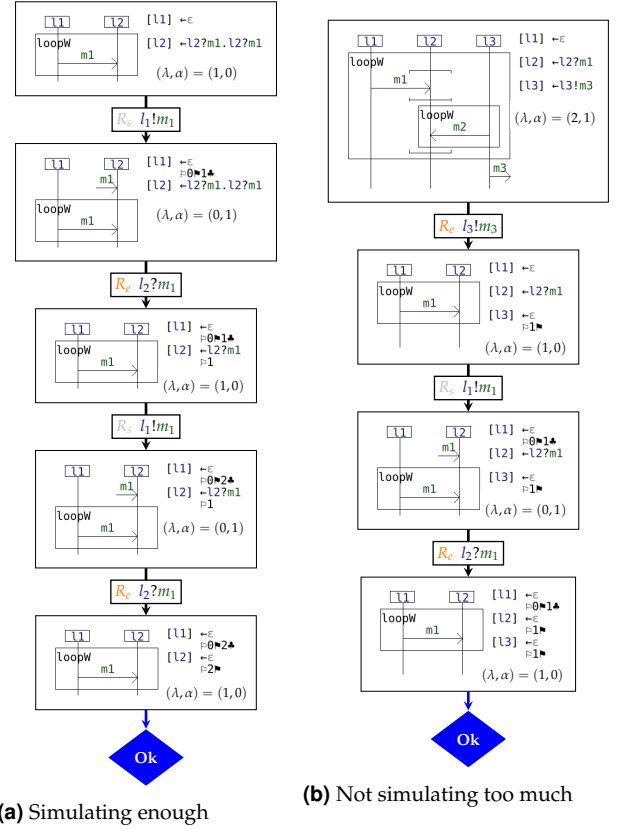
constructs of the interaction language, in combination with certain architectures of observation, may yield to situations where some correct slices are misidentified (i.e. an *Inconc* is returned instead of a *Pass*). We provide three such examples in Fig.11.

Let us consider the example from Fig.11a. Because of the use of the parallel loop $loop_P$, several instances of $seq(l!m_1, l?m_2)$ can be executed in parallel. In particular the trace $l!m_1.l!m_1.l!m_1.l?m_2.l?m_2.l?m_2$ is thus specified by the interaction. Hence, $l?m_2.l?m_2.l?m_2$ is a slice of an accepted behavior. However, in order to recognize this slice, the action $l!m_1$ should be simulated three times consecutively in order to reproduce the prefix missing from the slice. Using the criterion from Sec.5, we can only execute the action once because the maximum loop depth is 1 (it would be the same if we used the total number of loops). Hence, as illustrated on Fig.11a, the analysis fails. This problem is inherent to the $loop_P$ construct as long as the content of the loop (the sub-interaction within it) may express a behavior which contains several distinct actions (here $l!m_1$ and $l?m_2$).

The $loop_W$ construct is more forgiving in the general case. In particular it may not pose any problem when analyzing classical multi-traces (defined up to the discrete partition, as illustrated on Fig.10a). However, if distinct lifelines appearing in the sub-interaction underneath a $loop_W$ are co-localized it may pose a problem, as illustrated with the example from Fig.11b. Here the fact that l_1 and l_2

(a) Example with a *loop_P* construct.

(b) Example with a *loop_W* construct under a specific architecture (l_1 and l_2 co-localized).

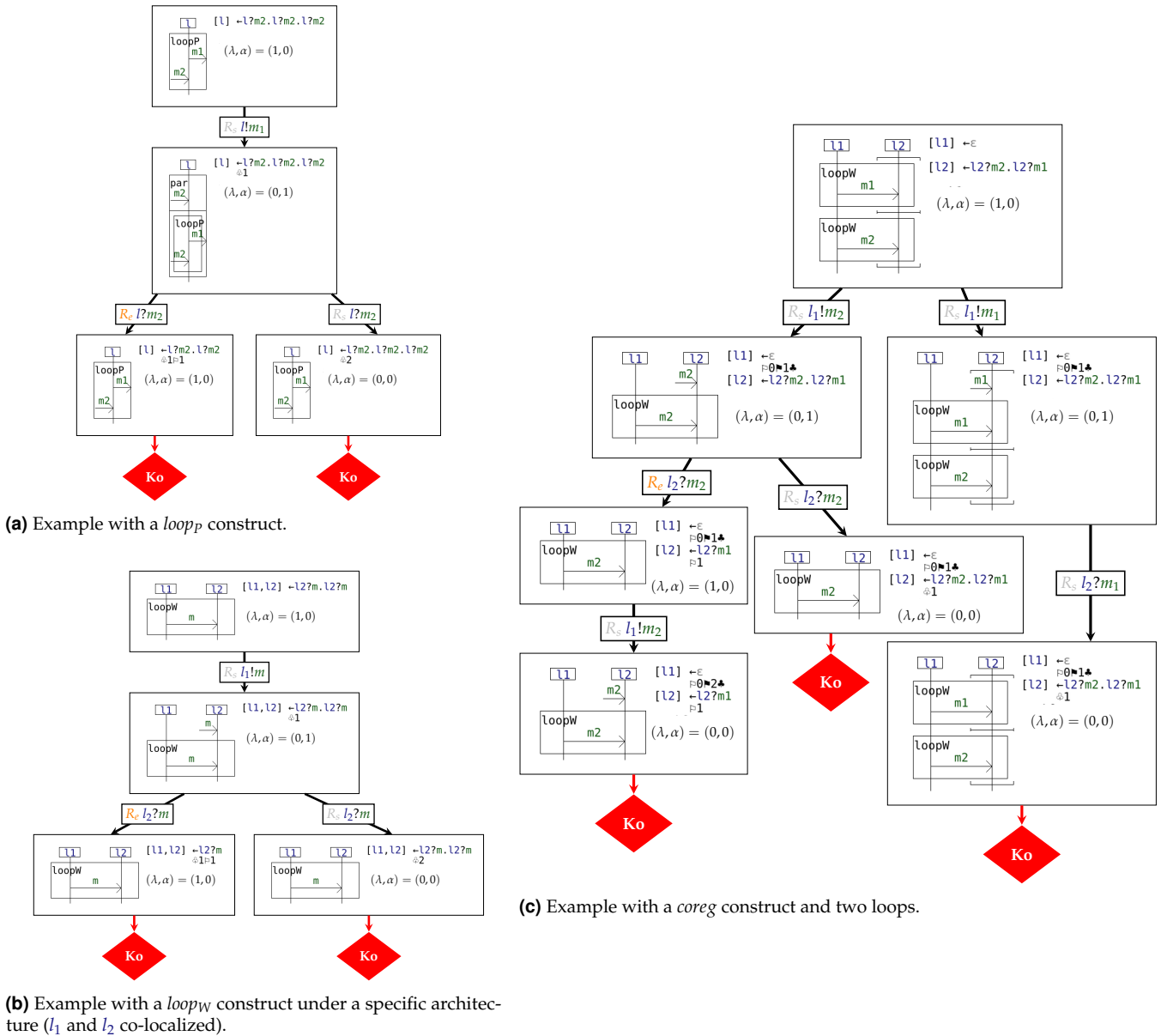**(c)** Example with a *coreg* construct and two loops.

Figure 11 Examples where the proposed criterion does not allow enough simulation to recognise correct slices.

are co-localized prevents the algorithm from alternating between simulation steps and execution steps (as it is done in Fig.10a) because simulation steps can only be taken outside the period of observation on a given co-localization.

The example from Fig.11c shows a similar problem. Due to the presence of the co-region, lifeline l_2 may receive incoming m_1 and m_2 messages in any order. However, lifeline l_1 must emit all the m_1 messages before it can emit the first m_2 message. In the specific multi-trace which we analyze on Fig.11c, l_2 receives one message m_2 and then one m_1. Because the maximum loop depth is 1, we can only simulate one emission consecutively. If we simulate $l_1!m_1$ first then we can't do anything afterwards because m_1 can't be received by l_2 yet and we cannot simulate $l_1!m_2$ due to the limitation on loops. If we simulate $l_1!m_2$

first then, due to transformations of the interaction (which specifies that l_1 must emit all the m_1 messages before it can emit the first m_2 message), we can not simulate $l_1!m_1$ afterwards and hence the analysis fails. In order for this specific analysis to succeed we would need to be able to instantiated 2 loops.

Let us remark that taking the total number of loops instead of the maximum depth of loops would help for this specific example (as the total number of loops is 2) but would not help in all cases. For instance, if a multi-trace μ with $\mu_{|\{l_1\}} = \varepsilon$ and $\mu_{|\{l_2\}} = l_2?m_2.l_2?m_2.l_2?m_1$ is analyzed, instantiating 3 loops is required.

5.5. Experimental assessment
We present a small experimental assessment of an implementation of our algorithm parameterized with the
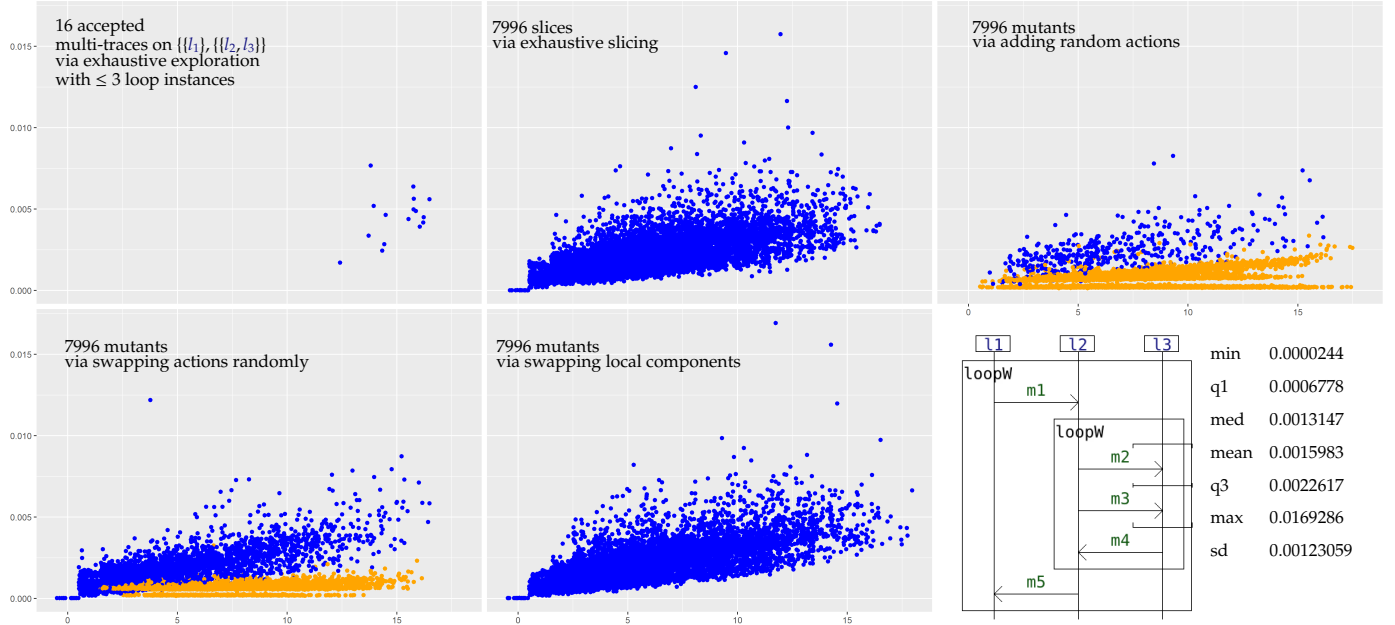
(a) Example with shallow but exhaustive exploration and exhaustive slicing.
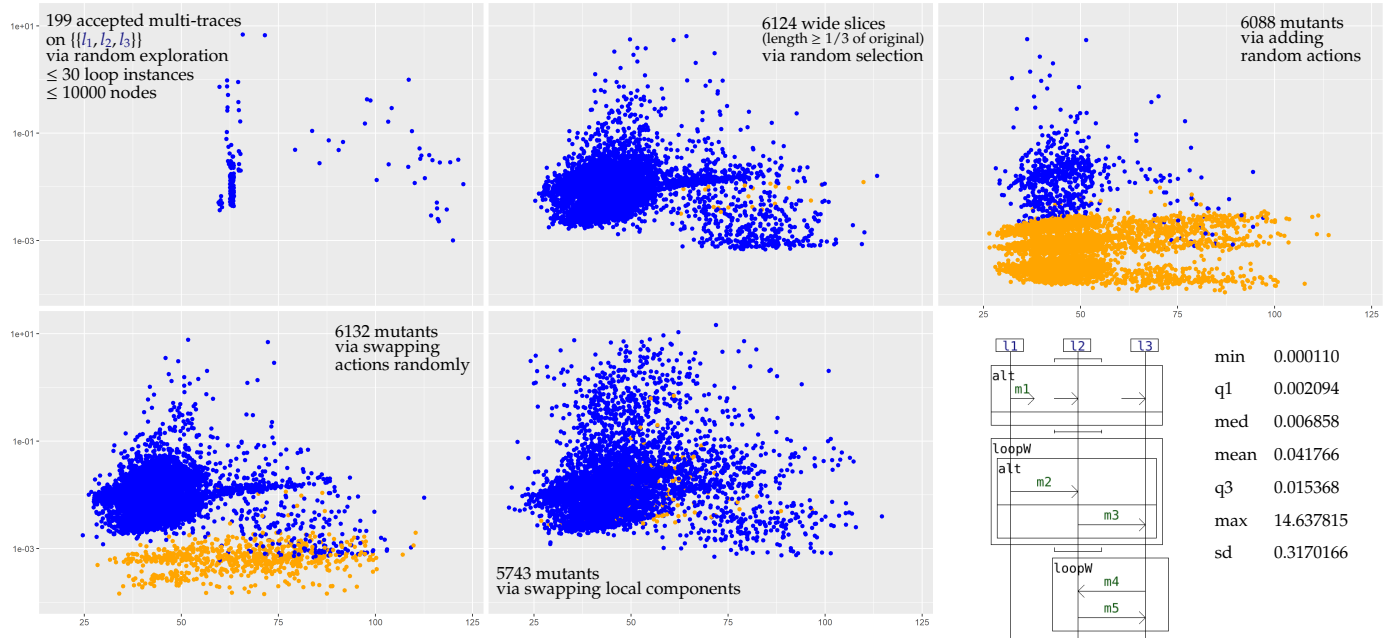
(b) Example with partial and random but in depth exploration and random selection of wide slices (length ≥ 1/3 of original). Time in log scale.

Figure 12 Some experiments (length and time resp. in x and y axes, blue and orange representing *Pass* and *Inconc* verdicts).

criterion shown in Sec.5.1. The results of those experiments are summarized on Fig.12. We exploit two interactions and for each one of them which we denote by i:

1. we generate a finite set $T(i) \subseteq \sigma_C(i)$ of accepted multi-traces given a certain partition C of lifelines.

2. for every accepted multi-trace μ, we generate a set of slices. Hence, we obtain a certain $S(i) \subseteq \widetilde{T(i)}$.

3. we then generate three sets of mutants from those slices $M_{sa}(i)$, $M_{sc}(i)$, and $M_{ia}(i)$. Those sets of mutants are obtained from $S(i)$ as follows:

   - $M_{sa}(i)$ is obtained by swapping the relative positions of actions within a local trace. For instance, given $\mu = (l_1!m_1.l_1!m_2, l_2?m_1.l_2?m_2)$, $\mu' = (l_1!m_2.l_1!m_1, l_2?m_1.l_2?m_2)$ is one such mutant. Here, we have swapped the positions of $l_1!m_1$ and $l_1!m_2$ on local trace $\mu_{|\{l_1\}}$.
   - $M_{sc}(i)$ is obtained by swapping local traces between two distinct slices related to the same component. For instance, given $\mu_1 = (l_1!m_1, l_2?m_1)$ and $\mu_2 = (l_1!m_2, l_2?m_2)$, $\mu' = (l_1!m_1, l_2?m_1)$ is one such mutants.
   - $M_{ia}(i)$ is obtained by inserting random actions. For instance, given $\mu = (l_1!m_1, l_2?m_1)$, $\mu' = (l_1!m_1.l_1!m_3, l_2?m_1)$ is one such mutant.

The generated multi-trace sets i.e. $T(i)$, $S(i)$, $M_{sa}(i)$, $M_{sc}(i)$, and $M_{ia}(i)$ are used as input to feed our analysis algorithm. The median times (based on 5 tries) required to analyze each of them are plotted on Fig.12. Each point corresponds to a multi-trace, with on the x axis its length and on the y axis its median analysis time. This time is in seconds, and for Fig.12b, we use a logarithmic scale. The color of the point corresponds to the verdict of the analysis: blue and orange respectively denote *Pass* and *Inconc*.

For each interaction example (i.e. i_1 on Fig.12a and i_2 on Fig.12b), the 5 plots correspond to the five sets $T(i)$, $S(i)$, $M_{sa}(i)$, $M_{sc}(i)$, and $M_{ia}(i)$. Legends written on each plot describe how the corresponding multi-traces have been generated. The corresponding interaction diagram is drawn on the right of each figure, and statistics on the analysis time are given.

The experiments were performed using an Intel(R) Core(TM)i5-6360U CPU (2GHz) with 8 GB of RAM. All the material required to reproduce them is publicly available in (Mahe 2023b).

We can at first notice that the algorithm's performances are highly dependent on the nature of the input interaction. The more the interaction offers branching choices (loops, alternatives) and possible interleavings (weak sequencing and interleaving), the greater can the size of graph $\mathbb{G}$ be, with, as a consequence, worse performances. While example i_1 (Fig.12a) is rather sequential, this is not the case for i_2 (Fig.12a).

The dependence w.r.t. the size of the input multi-trace appears linear for interaction i_1 (Fig.12a) and is less noticeable for interaction i_2 (Fig.12b). The analysis time is much more dependent on the structure of the input multi-trace w.r.t. the interaction rather than on its size.

Slices   Our criterion is capable of correctly identifying slices in most cases, as illustrated with $S(i_1)$ and $S(i_2)$ (top middle plot in both Fig.12a and Fig.12b). $S(i_1)$ contains all the possible slices (7996) of the multi-traces from $T(i_1)$, and all of them are correctly identified (drawn in blue color). Even though $S(i_2)$ contains slices of much larger multi-traces (up to 30 instances of the loops of i_2), most of those are still correctly identified. The few *Inconc* verdicts correspond to cases where the criterion does not allow enough simulation steps, as illustrated in Sec.5.4.

Mutants by adding random actions   In most cases, when we add a random action to a multi-trace which is conform to a specification, it becomes non-conform. This is reflected on the top right plots of both Fig.12a and Fig.12b by the fact that most multi-trace analyses are inconclusive (*Inconc* verdict drawn in orange color). Let us recall that because of the limitations of our criterion illustrated in Sec.5.4, we cannot return a *Fail* verdict. In terms of performances, in the case of $M_{ia}(i_1)$ and $M_{ia}(i_2)$, we can obtain the *Inconc* verdicts more quickly on average than the *Pass* verdicts. This is because we use some optimizations on the exploration of the graph (we use local analyses to cut parts of $\mathbb{G}$).

Mutants by swapping actions   Suppose the specifying interaction allows many possible interleavings of actions. In that case, swapping the positions of actions of a conform slice is more likely to yield a new multi-trace which is also conform. This is reflected on the bottom left plots of Fig.12a and Fig.12b, where we can see that those mutants are conform in many cases.

Mutants by swapping local components   This family of mutants is quite interesting given that for any such mutant obtained from two conform slices, the local components of the mutant are still all locally conform to the specification. This makes techniques such as using local analyses to reveal non-conformities useless. If the mutant is non-conform, then only a global analysis - i.e. matching the multi-trace to an accepted global scenario - can identify this non-conformity. In any case, most of those mutants are conform because we only consider a labeled language (i.e. no message passing and no value passing). With message passing, there are likely mismatches between the messages that are passed between non-co-localized lifelines. Still, in our purely labeled framework, some inconclusive verdicts are present for interaction i_2 (Fig.12b).

Concluding remarks   Let us remark that the presence of inconclusiveness is related to the fact that we have to identify strict slices of multi-traces and to the expressiveness

of the specification language (co-regions, parallel loops etc.) and the trace language (multi-traces with co-localized lifelines). If we restrict the prerequisites of the analysis, we can use algorithms that do not return *Inconc* verdicts. The algorithms from (Mahe et al. 2020) and (Mahe et al. 2021) can respectively identify full accepted global traces and full accepted multi-traces. In (Mahe et al. 2023), we define an algorithm which, instead of using simulation, applies a lifeline removal operation on no-longer-observed lifelines in order to identify multi-prefixes of multi-traces defined on the discrete partition. Multi-prefixes are multi-slices in which missing actions are only located at the end of the local trace components of multi-traces.

For the algorithm based on simulation, these few experiments show that our tool can analyze representative multi-traces with respect to interactions with reasonable performances. In the presence of an inconclusive verdict, the user may:

- revert to using one of the other aforementioned algorithms, if their prerequisites are valid (e.g. synchronization at the start etc.),
- start again with a more liberal criterion,
- or analyze the multi-trace by hand.

One such more liberal criterion would consist in having the measure $\lambda = \beta(i)$ being multiplied by the size of the multi-trace $|\mu|$ without resetting the measure whenever rule R_e applies. In essence, this amounts to instantiating the (nested) loops as many times as the number of actions in the multi-trace. With this criterion we would be able to recognize all the examples from Fig.11. However, this criterion can produce exponentially larger analysis graphs $\mathbb{G}$, with a strong impact on the algorithm's performance. This increases the interest in looking into more selective criteria, such as the one which we proposed, at the expense of the completeness of the analysis.

6. Related works

The aim of Runtime Verification (RV) (Bartocci et al. 2018; Sánchez et al. 2019) is to test the conformity of an implemented system against a formal specification which may define a set of accepted and/or unwanted behaviors. To do so, traces - characterizing system executions - are collected by an instrumentation of the system and then confronted to the formal specification. If one such trace deviates from the specification (i.e. does not characterize an accepted behavior or do characterize an unwanted behavior) then the tester has found a bug in the implemented system. RV techniques include offline and online RV. In online approaches the confrontation to the formal specification takes place at the same time the system is being monitored via the instrumentation. This has the advantage of being able to monitor live reactive system as they are being executed (and expressing behaviors that can be extended arbitrarily many times). However online RV is quite constrained by requirements on the efficiency

of the monitoring algorithm. Indeed, observed events must be analyzed quickly enough so that they don't stack and cause a memory overflow. By contrast, in offline approaches, a trace is collected in a first step and then confronted to the specification. As a result, only finite traces can be analyzed but Offline Runtime Verification (ORV) has fewer constraints than its online counterpart. In the following, we only consider ORV.

6.1. RV for DS

In the literature, most approaches for applying ORV to DS can be categorized as either **(1)** based on global re-ordering or **(2)** based on a synthesis of local analyses. Fig.13a and Fig.13b respectively describe those two kinds of approaches.

(1) ORV approaches based on global reordering are often found for solving the Oracle problem in Model Based Testing (MBT). Despite the fact that MBT is supposed to be used at the design phase while RV more naturally concerns the operational phase of a system, algorithms to solve the oracle problem in MBT are technically very similar to offline RV algorithms: they both consist in analyzing an execution to detect non-conformance to a specification. In such MBT approaches, DS behaviors can be modeled using e.g. Input/Output Transition Systems (IOTS) (Hierons et al. 2008, 2011) or Communicating Sequential Processes (CSP) (Cavalcanti et al. 2011). In those works, local observations are intertwined to associate them with global traces that can be analyzed w.r.t. models. This is described on Fig.13a as "reordering". Those approaches however require local observations to be synchronized, based on the states in which each of the logging processes terminate (e.g. based on quiescence states (Hierons et al. 2008), termination/deadlocks (Cavalcanti et al. 2011) or pre-specified synchronization points (Hierons et al. 2011)). The works (Dan & Hierons 2014; Nguyen et al. 2012) focus on verifying distributed executions against models of interaction (While (Dan & Hierons 2014) concern MSC, (Nguyen et al. 2012) considers choreographic languages). Similarly to the MBT approaches, they rely on synchronization hypotheses and on reconstructing a global trace by ordering events occurring at the distributed interfaces (by exploiting the observational power of testers (Dan & Hierons 2014) or timestamp information assuming clock synchronization (Nguyen et al. 2012)). In both MBT and RV inspired approaches, synchronization hypotheses restrict the conditions under which DS can be verified.

(2) ORV approaches based on a synthesis of local analyses are often found whenever local monitors are synthesized from a global property. Using temporal logic properties as specifications for RV of DS has been widely studied. In particular, using (variants of) the Linear Temporal Logic (LTL). For example, (Sen et al. 2004) extends the LTL in a framework where formulas relate to subsystems and what they know about the other sub-systems (e.g. in which local states they are). There is a collection of decentralized observers sharing information about the
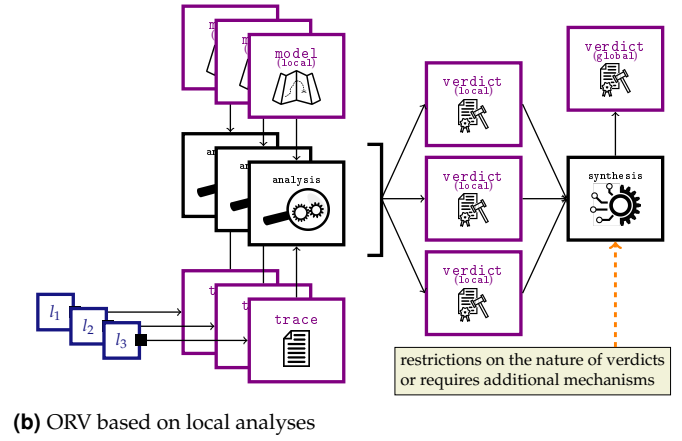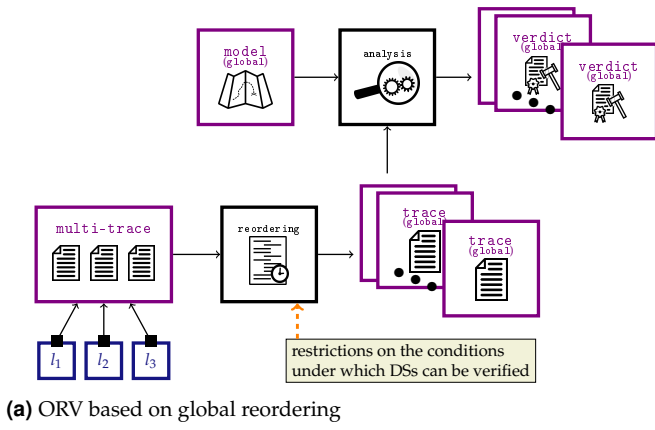
(a) ORV based on global reordering

(b) ORV based on local analyses

Figure 13 ORV approaches from the literature

sub-system executions affecting the validity of the formula. In other works (Bauer & Falcone 2016; El-Hokayem & Falcone 2017), the reference properties are expressed at the (global) system level. A collection of decentralized observers is generated from such properties, using LTL formula rewriting, so that there is no need for a global verifier gathering all information on the execution of the system. Approaches based on logics derived from modal μ-calculus (e.g. safety Hennessy-Milner Logic sHML) also implement synthesis of monitors. In (Attard & Francalanza 2017; Attard et al. 2021), instrumentation and monitors for programs running on the Erlang Virtual Machine are synthesized from properties written in sHML. Behavioral models can also be used in such approaches. For instance (Bocchi et al. 2017) proposes local online RV against projections of multiparty session types satisfying some conditions that enforce intended global behaviors (possibly combined with analysis through a global safe router). More generally, the fact that every local behavior is conform w.r.t. its corresponding local specification, does not generally implies that the global behavior is conform. Hence, the synthesis step described on Fig.13b generally requires additional mechanisms. In most cases this consists in having local monitors communicate with one another (e.g. in (Bauer & Falcone 2016)). This in return requires a very specific instrumentation which goes well beyond simply logging traces.

(Falcone et al. 2021) provides a taxonomy of RV tools. Specification languages which are most commonly found and fitted with tools derive from Temporal Logic. Unlike such logics, behavioral models of interactions, which are particularly adapted for specifying DS, are more rarely used. The works (Dan & Hierons 2014; Nguyen et al. 2012; Bocchi et al. 2017; Inçki & Ari 2018) focus on verifying distributed executions against models of interaction (While (Dan & Hierons 2014; Inçki & Ari 2018) concern MSC, (Nguyen et al. 2012) considers choreographic languages, (Bocchi et al. 2017) session types and (Ancona et al. 2018) trace expressions).

6.2. RV under partial observation

In this context, *partial observation* signifies that the multi-trace logged by the instrumentation does not characterize the entire execution of the DS. More concretely, some events may be missing from the multi-trace w.r.t. an ideal multi-trace which would have been observed with ideal conditions of observation. The notion of *multi-trace slice* from Sec.2.2 proposes a specific definition of partial observation, where events may be missing at the beginning and/or the end of each local trace component of the multi-trace (independently).

In the literature, ORV approaches for DS which are tolerant to partial observation are rare. In (Ancona et al. 2018), the authors are interested in generating monitors for distributed RV and in particular, how those monitors can be adapted for a specific definition of partial observation. Here, messages are exchanged via channels which are associated to an observability likelihood. (Ancona et al. 2018) uses trace expressions as specifications and proposes transformations that can adapt those expressions to partial observation by removing or making optional a number of identified unobservable events.

The fact that such missing actions may be located anywhere in a globally sequential behavior (as illustrated in Fig.6a) is particularly challenging for ORV. Indeed, if we base ORV on the recognition of a global behavior against a behavioral model (e.g. choreographies (Nguyen et al. 2012), MSC (Dan & Hierons 2014) or interactions (Mahe et al. 2020)) then the observed behavior must match a behavior that can be run through the model. This is not possible in the presence of missing actions.

In (Mahe et al. 2023), a partial solution to this problem is given via the use of a lifeline removal operation which is applied on no-longer-observed sub-systems. During the analysis, once a local trace component is entirely re-enacted, the behavioral model is simplified by removing all behavior that concerns the no-longer-observed component. This enables the analysis to be pursued even if the observation on some components has ceased too early.

6.3. Position of the contribution

We propose an approach for ORV which confronts observed multi-traces to positive requirements in the form of behavioral models. Our approach does away with synchronization hypotheses on the beginning and the end of observation in between distant observers. As a result, the observed multi-traces can be slices of multi-traces that fully characterize executions of the DS (as defined in Sec.2). We use interactions (Mahe et al. 2022) as formal specifications.

Neither monitorability nor implementability are in the scope of this paper. Implementability refers to whether or not a DS specified by an interaction can be implemented concretely e.g. for HMSC in (Lohrey 2003; Alur et al. 2001). DS specified by interactions which are not implementable cannot consistently produce traces which are conform to the specification. Monitorability is generally associated to properties written in temporal logic or Hennesy-Milner logic and refers to whether or not is possible to validate or invalidate the satisfaction of a property via monitoring (e.g. see partial monitorability in (Aceto et al. 2019)).

We propose an algorithm which checks whether or not an input multi-trace is a slice of a multi-trace that belong to the semantics of an input interaction. Instrumentation or recording methodology, which covers the manner with which such multi-traces can be obtained, is out of the scope of this paper. Because we do not have strong hypotheses on the synchronization of observers, a very simple and lightweight instrumentation could be used. Depending on the usecase and its level of abstraction, we could either use logs printed by the different machines (which correspond to co-localizations) or capture and filter network traffic in and out of those machines e.g. with the pcap library (McCanne 2011) or Wireshark (Wireshark 2023) as demonstrated in e.g. (Fowler & Hammel 2014).

7. The tool

Our tool HIBOU (for Holistic Interaction Behavioral Oracle Utility)[3] provides utilities for drawing, manipulating and exploiting interaction. The version which we describe in this paper is version 0.8.4. Its code (written using Rust) is hosted on GitHub in (Mahe 2022).

7.1. Encoding of interactions and traces

The specification language of our tool covers different notions:

- system signatures are defined in ".hsf" (hibou signature file) files as illustrated on Fig.14. In those files, we declare the set of messages M and lifelines L that constitute the signature of the Distributed System.
- interaction terms are specified in ".hif" (hibou interaction file) files as illustrated on Fig.15.
- multi-traces are encoded in ".htf" (hibou trace file) files as illustrated on Fig.16

---

[3] The word "hibou" stands for owl in French.

```
1  @message{
2    m1;m2;m3;m4;m5
3  }
4  @lifeline{
5    l1;l2;l3
6  }
```

Figure 14 Declaration of a signature.

On Fig.15 we provide the encoding of the interaction model that is drawn on Fig.2a. This textual format (found in ".hif" files) is similar to the notations used in Sec.3. The basic building blocks are the empty interaction $\varnothing$ encoded as "o" (lowercase letter o) and communication actions. For actions, we use notations inspired by WebSequenceDiagrams (WebSequenceDiagrams 2022) / PlantUML (PlantUML 2022):

- instead of $l_1!m$ we write `l1 -- m ->|`
- instead of $l_1?m$ we write `m -> l1`
- $strict(l_1!m, seq(l_2?m, l_3?m))$ becomes `l1 -- m ->(l1,l2)`

The *strict*, *seq*, *par*, *coreg$_r$*, *alt*, *loop$_S$*, *loop$_W$* and *loop$_P$* constructors directly match text labels in the encoding (we use parentheses to specify concurrent regions e.g. `coreg(l2)` for *coreg$_{\{l_2\}}$*) and we use parentheses to enclose sub-interactions. For any associative operator, we allow *n*-ary notations. For instance, $seq(i_1, i_2, i_3)$ is interpreted as $seq(i_1, seq(i_2, i_3))$.

```
1  seq(
2    coreg(l2)(
3      alt(
4        l1 -- m1 -> (l2,l3),
5        o
6      ),
7      loopW(
8        alt(
9          l1 -- m2 -> l2,
10         l2 -- m3 -> l3
11    ) )
12   ),
13   loopP(
14     seq(
15       l3 -- m4 -> l2,
16       l2 -- m5 -> l3
17 ) ) )
```

Figure 15 Encoding of the interaction from Fig.2

The encoding of multi-traces (within ".htf" files) is straightforward, as illustrated on Fig.16. Co-localizations are declared between square brackets:

- either in plain text or via a keyword:
- `[#all]` signifies that all lifelines are in this co-localization and thus we have a global trace
- `[#any]` signifies that the lifelines appearing on the actions of the trace component are taken into account to define the co-localization

– if a lifeline from *L* is left without any co-localization that contains it, then a dedicated co-localization with an empty trace component is created for it

```
1  [#any] l1!m1.l2?m1.l2?m4;
2  [l3]    l3?m1.l3!m4
```

Figure 16 Encoding of the multi-trace from Fig.1c

7.2. Command line interface

The HIBOU tool takes the form of a Command Line Interface which includes the following commands:

– "`hibou draw <.hsf file> <.hif file>`" draws an interaction in a graphical form (which we have used in this paper);
– "`hibou explore <.hsf file> <.hif file> <.hcf file>?`" explores the semantics of an interaction i.e. it computes and may display parts of the execution tree of that interaction as well as generate accepted multi-traces;
– "`hibou analyze <.hsf file> <.hif file> <.htf file> <.hcf file>?`" analyzes a multi-trace against an interaction i.e. it computes and may display parts of an analysis graph related to that analysis and returns a global verdict.

For the "`explore`" and "`analyze`" commands we may also provide a ".hcf" (hibou configuration file) file to further configure the exploration or analysis process (so as to replace the default configuration).

7.3. Semantics exploration & heuristics

The operational semantics from Sec.3.2 is implemented in the tool which enables exploring execution trees for any interaction via the "`hibou explore`" command. This exploration can be configured via a ".hcf" file, within a `@explore_option` section, as illustrated on Fig.17.

```
1  @explore_option{
2    loggers = [graphic[svg,vertical],
3              tracegen[generation = exact,
4                partition={(l1,l2),(l3)}]
5            ];
6    strategy = HCS;
7    filters = [max_depth = 35,
8              max_loop_depth = 4,
9              max_node_number = 250];
10   priorities = random
11 }
```

Figure 17 An example configuration for an exploration

From each new node of the tree, immediately executable actions are determined and their execution scheduled for the exploration. This scheduling order is by default the lexicographic order of their positions (hence actions "at the top" are prioritized over those "at the bottom"). We

can change this by setting either a random order or some priorities (e.g. prioritizing emissions over receptions or actions outside or within loops). The exploration of the tree is then performed according to a certain (deterministic) search strategy, which can be Breadth First, Depth First or a High Coverage Search that favors paths sharing fewer common prefixes. As interactions may contain loops, an exploration without constraints would not terminate. With HIBOU, we can set some limits on the exploration (e.g. a maximum depth, a maximum number of loops or of nodes).

This process can be observed by loggers, which can write outputs describing the exploration of the execution tree. A graphic logger provides a graphical representation of the analysis graph. It is enabled via `loggers = [graphic [svg,vertical]]`. We can configure its output format (svg or png) and we can decide whether the drawing is drawn from top to bottom (vertical) or left to right (horizontal). A trace generation logger can be set up via the `tracegen[ generation = exact,partition=..]`. For each path in the tree it may generate a ".htf" file containing a multi-trace which corresponds to this path. The keyword `exact` signifies that only exactly accepted traces will be generated (as opposed to using `prefix` for generating all prefixes or `terminal` for generating a trace only on terminal nodes of the explored tree). `partition=..` is used to specify the partition of lifelines on which the multi-traces are generated (we can use the `discrete` and `trivial` keywords for the C_d and C_t partitions).

7.4. Multi-trace analysis

HIBOU implements several configurable algorithms for analyzing multi-traces w.r.t. interactions. This process corresponds to the "`hibou analyze`" command and can be configured via a `@analyze_option` section as illustrated on Fig.18. Most options are defined in common with those of the exploration process. However, some are specific to analyses.

```
1  @analyze_option{
2    loggers = [graphic[svg]];
3    analysis_kind = simulate
4                  [before = true,
5                   loop max depth,
6                   reset = true,
7                   multiply = false,
8                   act num = 10];
9    strategy = DFS;
10   priorities = [simu = -1];
11   goal = WeakPass
12 }
```

Figure 18 An example configuration for an analysis

"`analysis_kind`" can be set to specify which analysis algorithm should be used. Among others it proposes:

– "`analysis_kind = accept`" which identifies exactly accepted multi-traces and corresponds to the algorithm from (Mahe et al. 2021). It returns *Pass* if this is the case and *Fail* otherwise.

- "`analysis_kind = prefix`" which identifies behaviors which are projections of prefixes of accepted global traces. It is adapted from (Mahe et al. 2021) and returns *Pass* if the behavior is exactly accepted, *WeakPass* if it corresponds to such a prefix and *Fail* otherwise.
- "`analysis_kind = eliminate`" which identifies prefixes of accepted multi-traces (events missing at the end of local components) defined over the discrete partition. Prefixes in the sense of multi-traces are not necessarily projections of prefixes of global traces. As such, to underline this difference, we may call them multi-prefixes. It corresponds to the algorithm from (Mahe et al. 2023) which is based on the use of a lifeline removal operation which is applied on no-longer-observed sub-systems. This algorithm returns *Pass* if the behavior is exactly accepted, *WeakPass* if it corresponds to a multi-prefix and *Fail* otherwise.
- "`analysis_kind = simulate[...]`" which implements the algorithm from this paper. It can be configured with a number of options:
    - the "`loop`" option sets up a stopping criterion on the number of loops that can be instantiated in simulation. It can either correspond to the maximum number of loops in the interaction, the maximum depth of its nested loops or to a specific ad-hoc number.
    - the "`act`" option sets up a stopping criterion on the number of actions that can be executed in simulation. It can either correspond to the number of actions outside loops or a specific ad-hoc number.
    - if set, the "`before`" option activates simulation before the beginning of observation in addition to after the end of observation on local components. With this option we can recognize slices and without it we can only recognise multi-prefixes.
    - if set, the "`multiply`" option multiplies the criteria on loops and actions by the size of the multi-trace to analyze.
    - if set, the "`reset`" option makes so that the measure is reset after a step of execution (if not, then we have a set number of simulation steps for the whole analysis, independently of the use of R_e)

This algorithm may return a *Pass* or *WeakPass* if the multi-trace is exactly accepted or either a prefix or a slice of an accepted one or return *WeakFail* if this is not the case.

With the "`goal`" option, we can force the termination of the search once a sufficient verdict is found e.g. *WeakPass*.

8. Conclusion

In this paper, we propose a new ORV approach which can be adapted to various observation architectures and is tolerant to the absence of synchronization between local observers. Different sub-systems deployed on the same computer can be modeled by several co-localized lifelines. The notion of co-localization allows us to use the same approach to treat the analysis of both global traces (in the case where the system is centralized i.e. all sub-systems are deployed on the same machine and hence all corresponding lifelines are co-localized) and multi-traces (when the system is fully distributed i.e. all sub-systems are deployed on distinct machines and no two lifelines are co-localized). Moreover, our handling of partial observation allows taking into account situations where the executions of some sub-systems are not (or are partially) observable due to technical limitations, in particular related to missing observers or the absence of synchronization mechanisms between distant observers.

Multi-trace analysis combines steps of consumption of actions present in the multi-trace and simulation steps to guess missing (i.e. unobserved) actions. The simulation steps are controlled by a criterion to ensure that the analysis is performed in a finite time. At then end of the analysis, two verdicts may be emitted: *Pass* when a slice is recognized and *Inconc* if not. Because actions may be missing at the beginning of local traces and because interactions may include loops, further characterizing the inconclusive verdict may require arbitrarily many simulation steps and is thus not tractable in all generality. Since accepted multi-traces of an interaction are determined via an operational-style semantics, we can visualize follow-up interactions. Our approach being implemented into the HIBOU tool, this allows users to write and debug interactions at the design stage.

References

Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., & Lehtinen, K. (2019, jan). Adventures in monitorability: From branching to linear time and back again. *Proc. ACM Program. Lang.*, *3*(POPL). doi: 10.1145/3290365

Alur, R., Etessami, K., & Yannakakis, M. (2001). Realizability and verification of MSC graphs. In F. Orejas, P. G. Spirakis, & J. van Leeuwen (Eds.), *Automata, languages and programming, 28th international colloquium, ICALP 2001, crete, greece, july 8-12, 2001, proceedings* (Vol. 2076, pp. 797–808). Springer. doi: 10.1007/3-540-48224-5_65

Ancona, D., Ferrando, A., Franceschini, L., & Mascardi, V. (2018). Coping with bad agent interaction protocols when monitoring partially observable multiagent systems. In Y. Demazeau, B. An, J. Bajo, & A. Fernández-Caballero (Eds.), *Advances in practical applications of agents, multi-agent systems, and complexity: The paams collection* (pp. 59–71). Cham: Springer International Publishing.

Attard, D. P., Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., & Lehtinen, K. (2021). Better late than never or: Verifying asynchronous components at runtime. In K. Peters & T. A. C. Willemse (Eds.), *Formal techniques for distributed objects, components, and systems* (pp. 207–225). Cham: Springer International Publishing.

Attard, D. P., & Francalanza, A. (2017). Trace partitioning and local monitoring for asynchronous components. In A. Cimatti & M. Sirjani (Eds.), *Software engineering and formal methods* (pp. 219–235). Cham: Springer International Publishing.

Bannour, B., Gaston, C., & Servat, D. (2011). Eliciting unitary constraints from timed sequence diagram with symbolic techniques: Application to testing. In T. D. Thu & K. R. P. H. Leung (Eds.), *18th asia pacific software engineering conference, APSEC 2011, ho chi minh, vietnam, december 5-8, 2011* (pp. 219–226). IEEE Computer Society. doi: 10.1109/APSEC.2011.40

Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). Introduction to runtime verification. In E. Bartocci & Y. Falcone (Eds.), *Lectures on runtime verification - introductory and advanced topics* (Vol. 10457, pp. 1–33). Springer. doi: 10.1007/978-3-319-75632-5_1

Bauer, A., & Falcone, Y. (2016). Decentralised LTL monitoring. *Formal Methods Syst. Des.*, *48*(1-2), 46–93. doi: 10.1007/s10703-016-0253-8

Benharrat, N., Gaston, C., Hierons, R. M., Lapitre, A., & Le Gall, P. (2017). Constraint-based oracles for timed distributed systems. In N. Yevtushenko, A. R. Cavalli, & H. Yenigün (Eds.), *Testing software and systems* (pp. 276–292). Cham: Springer International Publishing.

Bocchi, L., Chen, T., Demangeon, R., Honda, K., & Yoshida, N. (2017). Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, *669*, 33–58. doi: 10.1016/j.tcs.2017.02.009

Cavalcanti, A., Gaudel, M., & Hierons, R. M. (2011). Conformance relations for distributed testing based on CSP. In B. Wolff & F. Zaïdi (Eds.), *Testing software and systems - 23rd IFIP WG 6.1 international conference, ICTSS 2011, paris, france, november 7-10, 2011. proceedings* (Vol. 7019, pp. 48–63). Springer. doi: 10.1007/978-3-642-24580-0_5

Dan, H., & Hierons, R. M. (2014). The oracle problem when testing from mscs. *Comput. J.*, *57*(7), 987–1001. doi: 10.1093/comjnl/bxt055

El-Hokayem, A., & Falcone, Y. (2017). Monitoring decentralized specifications. In T. Bultan & K. Sen (Eds.), *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, santa barbara, ca, usa, july 10 - 14, 2017* (pp. 125–135). ACM. doi: 10.1145/3092703.3092723

Engels, A., Mauw, S., & Reniers, M. (2002). A hierarchy of communication models for message sequence charts. *Science of Computer Programming*, *44*(3), 253-292. doi: 10.1016/S0167-6423(02)00022-9

Falcone, Y., Krstic, S., Reger, G., & Traytel, D. (2021).

A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, *23*(2), 255–284. doi: 10.1007/s10009-021-00609-z

Faria, J. P., & Paiva, A. C. R. (2016). A toolset for conformance testing against UML sequence diagrams based on event-driven colored petri nets. *Int. J. Softw. Tools Technol. Transf.*, *18*(3), 285–304. doi: 10.1007/s10009-014-0354-x

Fowler, C. A., & Hammel, R. J. (2014). Converting pcaps into weka mineable data. In *15th ieee/acis international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (snpd)* (p. 1-6). doi: 10.1109/SNPD.2014.6888681

Hierons, R. M., Merayo, M. G., & Núñez, M. (2008). Controllable test cases for the distributed test architecture. In S. D. Cha, J. Choi, M. Kim, I. Lee, & M. Viswanathan (Eds.), *Automated technology for verification and analysis, 6th international symposium, ATVA 2008, seoul, korea, october 20-23, 2008. proceedings* (Vol. 5311, pp. 201–215). Springer. doi: 10.1007/978-3-540-88387-6_16

Hierons, R. M., Merayo, M. G., & Núñez, M. (2011). Scenarios-based testing of systems with distributed ports. *Softw. Pract. Exp.*, *41*(10), 999–1026. doi: 10.1002/spe.1062

Inçki, K., & Ari, I. (2018). A novel runtime verification solution for iot systems. *IEEE Access*, *6*, 13501–13512. doi: 10.1109/ACCESS.2018.2813887

ITU. (2011, 08). *Message sequence chart (msc)*. itu.int/rec/T-REC-Z.120.

Katoen, J.-P., & Lambert, L. (1998). Pomsets for message sequence charts. In (pp. 197–207). (Formale Beschreibungstechniken fuer verteilte Systeme, 8. GI/ITG-Fachgespraech)

Knapp, A., & Mossakowski, T. (2017). UML Interactions Meet State Machines - An Institutional Approach. In F. Bonchi & B. König (Eds.), *7th conference on algebra and coalgebra in computer science (calco 2017)* (Vol. 72, pp. 15:1–15:15). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.CALCO.2017.15

Lamport, L. (2019). Time, clocks, and the ordering of events in a distributed system. In D. Malkhi (Ed.), *Concurrency: the works of leslie lamport* (pp. 179–196). ACM.

Lohrey, M. (2003, dec). Realizability of high-level message sequence charts: Closing the gaps. *Theor. Comput. Sci.*, *309*(1), 529–554. doi: 10.1016/j.tcs.2003.08.002

Mahe, E. (2022). *Hibou tool*. github.com/erwanM974/hibou_label.

Mahe, E. (2023a, 02). *Coq proof for the equivalence of the semantics with co-regions*. erwanm974.github.io/coq_interaction_semantics_equivalence_with_coregions/.

Mahe, E. (2023b, 03). *Experiments on the simulation-based algorithm implemented in hibou for recognising multi-trace slices*. github.com/erwanM974/hibou_simulation_usecases_for_slice_recognition.

Mahe, E., Bannour, B., Gaston, C., Lapitre, A., & Gall, P. L. (2023). Interaction-based offline runtime verification of

distributed systems. In H. Hojjat & E. Ábrahám (Eds.), *Fundamentals of software engineering* (pp. 88–103). Cham: Springer Nature Switzerland.

Mahe, E., Bannour, B., Gaston, C., Lapitre, A., & Le Gall, P. (2021). A small-step approach to multi-trace checking against interactions. In (p. 1815–1822). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3412841.3442054

Mahe, E., Gaston, C., & Gall, P. L. (2020). Revisiting semantics of interactions for trace validity analysis. In H. Wehrheim & J. Cabot (Eds.), *Fundamental approaches to software engineering - 23rd international conference, FASE 2020, held as part of the european joint conferences on theory and practice of software, ETAPS 2020, dublin, ireland, april 25-30, 2020, proceedings* (Vol. 12076, pp. 482–501). Springer. doi: 10.1007/978-3-030-45234-6_24

Mahe, E., Gaston, C., & Le Gall, P. (2022). Equivalence of denotational and operational semantics for interaction languages. In Y. Aït-Ameur & F. Crăciun (Eds.), *Theoretical aspects of software engineering* (pp. 113–130). Cham: Springer International Publishing.

Mauw, S., & Reniers, M. A. (1997). High-level message sequence charts. In *SDL '97 time for testing, sdl, MSC and trends - 8th international SDL forum, proceedings* (pp. 291–306). Elsevier.

McCanne, S. (2011). *libpcap: An architecture and optimization methodology for packet capture.*

Micskei, Z., & Waeselynck, H. (2011). The many meanings of uml 2 sequence diagrams: a survey. *Software & Systems Modeling*, *10*(4), 489–514.

Mouakher, I., Dhaou, F., & Attiogbé, J. C. (2022). Event-based semantics of UML 2.x concurrent sequence diagrams for formal verification. *J. Comput. Sci. Technol.*, *37*(1), 4–28. doi: 10.1007/s11390-021-1673-5

Nguyen, H. N., Poizat, P., & Zaïdi, F. (2012). Passive conformance testing of service choreographies. In S. Ossowski & P. Lecca (Eds.), *Proceedings of the ACM symposium on applied computing, SAC 2012, riva, trento, italy, march 26-30, 2012* (pp. 1528–1535). ACM. doi: 10.1145/2245276.2232020

OMG. (2017, 12). *Unified modeling language.* omg.org/spec/ UML/.

Parrow, J. (2001). An introduction to the π-calculus. In J. A. Bergstra, A. Ponse, & S. A. Smolka (Eds.), *Handbook of process algebra* (pp. 479–543). North-Holland / Elsevier.

PlantUML. (2022). plantuml.com/.

Plotkin, G. (2004, 07). A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, *60-61*, 17-139. doi: 10.1016/j.jlap.2004.05.001

Pratt, V. (1986, feb). Modeling concurrency with partial orders. *Int. J. Parallel Program.*, *15*(1), 33–71. doi: 10.1007/ BF01379149

Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bian-culli, D., Colombo, C., . . . Weiss, A. (2019). A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, *54*(3), 279–335. doi: 10.1007/s10703-019-00337-w

Sen, K., Vardhan, A., Agha, G., & Rosu, G. (2004). Efficient decentralized monitoring of safety in distributed systems. In A. Finkelstein, J. Estublier, & D. S. Rosenblum (Eds.), *26th international conference on software engineering (ICSE 2004), 23-28 may 2004, edinburgh, united kingdom* (pp. 418–427). IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICSE.2004.1317464 doi: 10.1109/ICSE.2004.1317464

WebSequenceDiagrams. (2022). websequencediagrams .com/.

Wireshark. (2023, 03). wireshark.org.

A. Details on the denotational semantics with co-regions

A.1. Operators on sets of traces

Definition 9 (Interleaving). *The set $t_1 \| t_2$ of interleavings of traces t_1 and t_2 is defined by:*

$$
\begin{aligned}
\varepsilon \| t_2 &= \{t_2\} \\
t_1 \| \varepsilon &= \{t_1\} \\
(a_1.t_1) \| (a_2.t_2) &= \{a_1.t \mid t \in t_1 \| (a_2.t_2)\} \cup \{a_2.t \mid t \in (a_1.t_1) \| t_2\}
\end{aligned}
$$

Property 1. *For any traces t_1 and t_2 we have:*

$$
t_1 \| t_2 = t_1 \times\!|_L t_2
$$

Proof. For proving $t_1 \| t_2 \subseteq t_1 \times\!|_L t_2$ we can reason by induction on trace t_1. See lemma `cond_seq_covers_interleaving_1` in Coq proof (Mahe 2023a).

For proving $t_1 \times\!|_L t_2 \subseteq t_1 \| t_2$ we can reason by induction on the conditions for $t \in t_1 \times\!|_L t_2$. See lemma `cond_seq_covers_interleaving_2` in Coq proof (Mahe 2023a).
□

Definition 10 (Conflict). *We define a conflict operator $\times$: $\mathbb{T}_{|L} \times L \to \{\top, \bot\}$ such that:*

$$
\varepsilon \times l = \bot \qquad and \qquad (a.t) \times l = (\theta(a) = l) \vee (t \times l)
$$

Property 2 (Conflict is conditional conflict with empty concurrent region). *For any traces t and lifeline l we have:*

$$
(t \times l) \Leftrightarrow (t \times\!|_\emptyset l)
$$

Proof. We can reason by induction on trace t. See lemma `no_condconf_no_lifelines_charac` in Coq proof (Mahe 2023a).
□

Definition 11 (Weak Sequencing). *The set $t_1 \times t_2$ of weak sequencings of traces t_1 and t_2 is defined by:*

$$
\begin{aligned}
\varepsilon \times t_2 &= \{t_2\} \\
t_1 \times \varepsilon &= \{t_1\} \\
(a_1.t_1) \times (a_2.t_2) &= \{a_1.t \mid t \in t_1 \times (a_2.t_2)\} \\
&\cup \{a_2.t \mid t \in (a_1.t_1) \times t_2, \neg(a_1.t_1 \times \theta(a_2))\}
\end{aligned}
$$

Property 3. *For any traces t_1 and t_2 we have:*

$$t_1 ⨳ t_2 = t_1 ⨳|_\emptyset t_2$$

Proof. For proving $t_1 ⨳ t_2 \subseteq t_1 ⨳|_\emptyset t_2$ we can reason by induction on trace t_1. See lemma `cond_seq_covers_weak_seq_1` in Coq proof (Mahe 2023a).

For proving $t_1 ⨳|_\emptyset t_2 \subseteq t_1 ⨳ t_2$ we can reason by induction on the conditions for $t \in t_1 ⨳|_\emptyset t_2$. See lemma `cond_seq_covers_weak_seq_2` in Coq proof (Mahe 2023a). □

A.2. Characterization of the pruning relations

Theorem 6 (An interaction that can't be pruned has conflicts on all accepted traces). *For any $L' \subseteq L$ and any $i \in \mathbb{I}$ we have:*

$$(i \overset{L'}{\not\hookrightarrow}) \Rightarrow (\forall t \in \rho(i),\ \exists l \in L',\ t ⨳|_\emptyset l)$$

Proof. At first we remark that as per Prop.2, $⨳|_\emptyset = ⨳$. We can then reason by induction on the structure of interaction i. See theorem `cannot_prune_characterisation_with_sem_de` in Coq proof (Mahe 2023a). □

Property 4 (An interaction which can be pruned on all lifelines accepts the empty trace). *For any $i, i' \in \mathbb{I}$ we have:*

$$(i \overset{L}{\hookrightarrow} i') \Rightarrow (\varepsilon \in \rho(i))$$

Proof. We can reason by induction on the structure of interaction i. See lemma `prune_all_equiv_accept_nil_1` in Coq proof (Mahe 2023a). □

Property 5 (An interaction which accepts the empty trace can be pruned on all lifelines). *For any $i \in \mathbb{I}$ we have:*

$$(\varepsilon \in \rho(i)) \Rightarrow (\exists i' \in \mathbb{I},\ i \overset{L}{\hookrightarrow} i')$$

Proof. We can reason by induction on the structure of interaction i. See lemma `prune_all_equiv_accept_nil_2` in Coq proof (Mahe 2023a). □

Property 6 (Prune does not introduce new behaviors). *For any $L' \subseteq L$, any $i, i' \in \mathbb{I}$ and any $t \in \mathbb{T}_{|L}$ we have:*

$$\left(\begin{array}{l} (i \overset{L'}{\hookrightarrow} i') \\ \wedge (t \in \rho(i')) \end{array} \right) \Rightarrow (t \in \rho(i))$$

Proof. We can reason by induction on the structure of interaction i. See lemma `prune_characterisation_with_sem_de_1` in Coq proof (Mahe 2023a). □

Property 7 (Prune conserves behaviors without conflicts). *For any $L' \subseteq L$, any $i, i' \in \mathbb{I}$ and any $t \in \mathbb{T}_{|L}$ we have:*

$$\left(\begin{array}{l} (i \overset{L'}{\hookrightarrow} i') \\ \wedge (t \in \rho(i)) \\ \wedge (\forall l \in L',\ \neg(t ⨳ l)) \end{array} \right) \Rightarrow (t \in \rho(i'))$$

Proof. We can reason by induction on the structure of interaction i and use Th.6. See lemma `prune_characterisation_with_sem_de_2` in Coq proof (Mahe 2023a). □

Property 8 (Prune removes conflicts). *For any $L' \subseteq L$, any $i, i' \in \mathbb{I}$ and any $t \in \mathbb{T}_{|L}$ we have:*

$$\left(\begin{array}{l} (i \overset{L'}{\hookrightarrow} i') \\ \wedge (t \in \rho(i')) \end{array} \right) \Rightarrow (\forall l \in L',\ \neg(t ⨳ l))$$

Proof. We can reason by induction on the conditions that make hypothesis $(i \overset{L'}{\hookrightarrow} i')$ valid. See lemma `prune_removes_conflicts` in Coq proof (Mahe 2023a). □

Theorem 7 (Prune characterization in denotational semantics). *For any $i, i' \in \mathbb{I}$, for any $L' \subseteq L$ we have:*

$$(i \overset{L'}{\hookrightarrow} i') \Rightarrow (\rho(i') = \{t \in \rho(i) \mid \forall l \in L',\ \neg(t ⨳|_\emptyset l)\})$$

Proof. At first we remark that as per Prop.2, $⨳|_\emptyset = ⨳$. Then we use Prop.6, Prop.7 and Prop.8. See theorem `prune_characterisation_with_sem_de` in Coq proof (Mahe 2023a). □

A.3. Characterization of the execution relation

Property 9. *For any $i, i' \in \mathbb{I}$, for any $t \in \mathbb{T}_{|L}$ and $a \in \mathbb{A}_{|L}$ we have:*

$$\left(\begin{array}{l} (i \overset{a}{\to} i') \\ \wedge (t \in \rho(i')) \end{array} \right) \Rightarrow (a.t \in \rho(i))$$

Proof. We can reason by induction on the conditions that make hypothesis $(i \overset{a}{\to} i')$ hold. This proof uses Prop.4, Prop.6, Prop.8. See lemma `execution_characterisation_with_sem_de_1` in Coq proof (Mahe 2023a). □

Property 10. *For any $i \in \mathbb{I}$, for any $t \in \mathbb{T}_{|L}$ and $a \in \mathbb{A}_{|L}$ we have:*

$$(a.t \in \rho(i)) \Rightarrow \left(\exists i' \in \mathbb{I},\ s.t. \left(\begin{array}{l} (i \overset{a}{\to} i') \\ \wedge (t \in \rho(i')) \end{array} \right) \right)$$

Proof. We can reason by induction on the structure of interaction i. This proof uses Prop.5, Prop.7. See lemma `execution_characterisation_with_sem_de_2` in Coq proof (Mahe 2023a). □

A.4. Equivalence of the semantics

Theorem 8. *For any $i \in \mathbb{I}$ we have:*

$$\sigma_{C_t}(i) \subseteq \rho(i)$$

Proof. We can reason by induction on a trace $t \in \sigma_{C_t}(i)$ and use Prop.4 for the case $t = \varepsilon$ and Prop.9 for the case $t = a.t'$. See theorem `op_implies_de` in Coq proof (Mahe 2023a). $\qed$

Theorem 9. *For any $i \in \mathbb{I}$ we have:*

$$\rho(i) \subseteq \sigma_{C_t}(i)$$

Proof. We can reason by induction on a trace $t \in \rho(i)$ and use Prop.5 for the case $t = \varepsilon$ and Prop.10 for the case $t = a.t'$. See theorem `de_implies_op` in Coq proof (Mahe 2023a). $\qed$

B. Details on the proposal criterion

In this paper we propose in Sec.5 a specific criterion to limit the amount of simulation steps which can be taken during the analysis of a multi-trace. This criterion is a concretization of the abstract criterion used in Sec.4 for defining the analysis algorithm.

B.1. Maximum loop depth

We set a maximum number of loops which can be instantiated during a continuous sequence of simulation steps. A loop is instantiated if an action within it is executed. For nested loops, we consider that the number of loops which are instantiated corresponds to the depth of the action within these nested loops. For instance, within $loop_S(seq(alt(a_1, o), loop_S(a_2)))$, a_1 is at a loop depth of 1 while a_2 is at a loop depth of 2. An action that is not within any loop is at loop depth 0. This notion of loop depth of a certain action is captured by the β function from Def.12, where, for any interaction $i \in \mathbb{I}$ and any one of its position $p \in pos(i)$, $\beta(i, p)$ gives the loop depth of the node (within the tree structure of the interaction) at position p (hence this is particularly true for actions).

At the beginning of a sequence of simulation steps, starting from a certain interaction i_0, this maximum number of loops is initialized at a value which corresponds to the maximum depth of nested loops in i_0 which we denote $\beta(i_0)$ (see Def.12). This allows every action occurring in the interaction to be simulated at least once in at least one path starting from this initial interaction i_0.

Definition 12 (Loop depth). *β defined over $\bigcup_{i \in \mathbb{I}}(\{i\} \times pos(i))$ is the function s.t.:*

- *for any $i \in \mathbb{I}$, $\beta(i, \varepsilon) = 0$*
- *for any $i_1, i_2 \in \mathbb{I}^2$ and any $p_1 \in pos(i_1), p_2 \in pos(i_2)$, for any $f \in \{str, alt\} \cup \bigcup_{r \subseteq L}\{coreg_r\}$:*
  - *$\beta(f(i_1, i_2), 1.p_1) = \beta(i_1, p_1)$*
  - *$\beta(f(i_1, i_2), 2.p_2) = \beta(i_2, p_2)$*

- *for any $i \in \mathbb{I}$, $p \in pos(i)$ and $k \in \{S\} \cup \bigcup_{r \subseteq L}\{C_r\}$, $\beta(loop_k(i), 1.p) = \beta(i, p) + 1$*

We then define: $\beta : \mathbb{I} \to \mathbb{N}$ s.t. $\forall\, i \in \mathbb{I}$:

$$\beta(i) = max_{p \in pos(i)}\beta(i, p)$$

We illustrate this with Fig.19 in which we explore the semantics of an initial interaction i_0 (i.e. its execution tree) with a limitation on the number of loops which can be instantiated. Here this limit is initialized at 2 which is the maximum loop depth of the initial interaction i.e. $\beta(i_0) = 2$. We can see that this allows all the actions occurring in the initial interaction to be expressed at least once in at least one path of the part of the tree that is explored. Within the context of multi-trace analysis with simulation, if the next action in the multi-trace which might be executed needs to be "unlocked" via performing some simulation steps, because this action must in any case appear in the interaction, it might then suffice to take advantage of this remark to parameterize simulation.
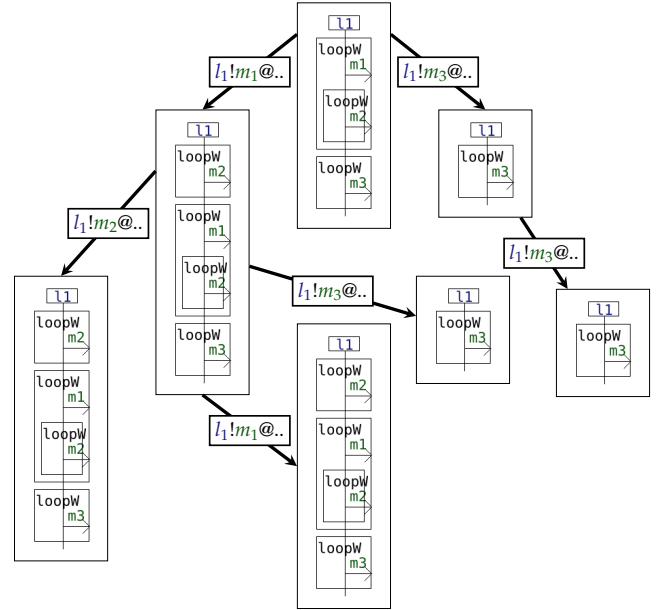

Figure 19 Semantics exploration (execution tree) with a limitation on the loop depth (here 2, which is the maximum loop depth of the initial interaction $\beta(i_0) = 2$).

B.2. Number of actions outside loops

Setting this limitation on the number of loops is sufficient to ensure termination of the algorithm because an interaction term being finite, there can only be a finite number of actions outside loops which may be simulated and because once such an action is simulated, it disappears from the follow-up interaction, in which the number of actions outside loops therefore diminishes by (at least) once. Yet by only considering the number of loop instantiation in our measure (i.e. by only considering λ), we do not have a strictly decreasing measure. Indeed, after steps where

an action outside a loop is simulated, λ stays the same. In order to reflect this decreasing number of actions, which appear in the interaction but not in λ, we introduce α as the number of actions outside loops, as defined in Def.13.

Definition 13 (Number of actions outside loops). *We define $\eta : \mathbb{I} \to \mathbb{N}$ as follows:*

- *$\eta(\varnothing) = 0$ and for any $a \in \mathbb{A}$, $\eta(a) = 1$*
- *for any $i_1, i_2 \in \mathbb{I}^2$:*
    - *for any $f \in \{strict\} \cup \bigcup_{r \subseteq L}\{coreg_r\}$, $\eta(f(i_1, i_2)) = \eta(i_1) + \eta(i_2)$*
    - *$\eta(alt(i_1, i_2)) = max(\eta(i_1), \eta(i_2))$*
- *for any $i \in \mathbb{I}$ and $k \in \{S\} \cup \bigcup_{r \subseteq L}\{C_r\}$, $\eta(loop_k(i)) = 0$*

By considering the tuple (λ, α) as a measure, we guarantee that successive simulation steps are bounded by a strictly decreasing measure (see Sec.5).