A small-step approach to multi-trace checking against interactions

Erwan Mahe, Boutheina Bannour, Christophe Gaston, Arnault Lapitre, Pascale Le Gall

A small-step approach to multi-trace checking against interactions

Erwan Mahe
Université Paris-Saclay, CentraleSupélec, MICS, Fr-91192 Gif-sur-Yvette Cedex

Boutheina Bannour
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Christophe Gaston
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Arnault Lapitre
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Pascale Le Gall
Université Paris-Saclay, CentraleSupélec, MICS, Fr-91192 Gif-sur-Yvette Cedex

ABSTRACT

Interaction models describe the exchange of messages between the different components of distributed systems. This paper presents an approach for checking the validity of multi-traces against interaction models. A multi-trace is a collection of traces (sequences of emissions and receptions), each representing a local view of the same global execution of the distributed system. We formally prove our approach, study its complexity, and implement it in a prototype tool.

KEYWORDS

interaction, small-step operational semantics, multi-trace analysis, distributed system

ACM Reference Format:
Erwan Mahe, Boutheina Bannour, Christophe Gaston, Arnault Lapitre, and Pascale Le Gall. 2021. A small-step approach to multi-trace checking against interactions. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3412841.3442054

1 INTRODUCTION

Context. A distributed system (DS) can be viewed as a collection of sub-systems, which are distributed over distinct physical locations and which communicate with each other by exchanging messages [13]. Analyzing the executions of DSs is a key problem to assess their correctness. However, the distributed nature of observations complicates this task. The absence of a global clock makes the classical notion of trace often too strong to represent DS executions. Indeed, a trace fully orders all events occurring in it while ordering events occurring on remote locations is often impossible. Therefore, multi-traces are better suited to model executions of DSs. A multi-trace is a collection of traces, one per sub-system, which represents the sequence of actions - emissions and receptions of messages - that have been observed at its interface. Contrary to traces, multi-traces do not strongly constrain orderings between actions occurring on different sub-systems. Our work is related to the general problem of the automatic analysis and debugging of DSs based on local logging of traces [2, 5, 15, 17, 18]. We are positioned at the intersection of two main issues: (1) that of tracking the causality of actions in traces [15, 17] based on the happened-before relation of Lamport [13] and (2) that of checking multi-traces against formal properties [2] or models [5, 18].

Contribution. In a model-based approach, we ground our analysis on interaction models (interactions for short) as references of intended DS executions. This kind of models - which include UML Sequence Diagrams [20], Message Sequence Charts [11], BPMN Choregraphies [19] among others - are widely used to specify DSs. In such models, DS executions are thought of as coordinations of message exchanges between multiple sub-systems. We consider interactions where the execution units are actions (the same as those constituting traces) and can be combined using operators of sequencing, choice, repetition and parallelism. This paper presents an approach to check the validity of multi-traces against interactions. Validity refers to the notion of being an *accepted* multi-trace, intuitively reflecting the fact of fully realizing *one* of the behaviours prescribed by the reference interaction, taking into account that interactions can be non-deterministic. We prove the correctness and discuss the complexity class of our method for analyzing multi-traces w.r.t. interaction semantics. Our present contribution extends a previous paper [16], where we proposed a small-step operational semantics for interactions, backed by an equivalent algebraic denotational semantics. As part of our contribution, we have also developed a prototype tool implementing the semantics and the algorithm for multi-trace analysis. This tool can render graphical representations detailing the steps taken by the analysis. Images of interactions in this paper have been adapted from its outputs.

Related work. Interactions have been extensively used to validate DSs using Test Case generation [3, 7, 14]. Much effort is spent on the generation of local test cases to mitigate the following problems: (1) "observability", i.e. the difficulty in inferring global executions from partial visions of message exchanges and (2) "controllability", i.e. the difficulty in determining when to apply stimuli in order to realize a targeted global execution. Our work falls within another domain which is Passive Testing [2, 18] (in which testers are only

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442054

observers), and discusses problems such as the Test Oracle Problem [8] (determining expected outputs w.r.t. given stimuli). Such a problem also falls into the domain of offline approaches to Runtime Verification [4, 23]. Both works [2, 18] have proposed approaches to check a set of local logs recorded in Service Oriented Systems. Authors in [2] propose a methodology to verify the conservation of invariants during the execution of the system. Both local and global invariants can be checked, although the latter is more costly in terms of computations. Our approach is different in that the reference for the analysis is not a correctness property but a model of interaction as in [8, 18]. Logics such as Linear Temporal Logic (LTL), are widely used in runtime verification to specify and verify requirements as logical properties. For DSs, either local properties are considered for synthesizing verifiers (as in the centralized case) in which case verification at a global level is difficult to reason about, or a global property is considered. In the latter case, either the property is transformed into decentralized verifiers and can lose meaning in the process, or all verifiers use the same global property, but they must be informed of other's local states [23]. There remains the possibility of coming back to the centralized case, but the accuracy of the global ordering of events using timestamping requires keeping the remote clocks synchronised [4]. In this perspective, models of interactions are well-suited to be used as a reference for correctness when analyzing DS executions. This is all the more relevant in cases where the temporal ordering of remote events is not feasible. [18] discusses passive testing against models of interactions expressed in the Chor [21] language. It differs from our approach in so far as: (1) Chor is less expressive than the interaction language we propose (particularly w.r.t. the absence of weak sequencing and the nature of loops), (2) [18] only handles synchronous communication between services, which cannot always describe accurately concrete implementations and (3) the local logs are not directly checked against the model but first pass through a synthesis step in which a global log is reconstituted based on timestamp information, and then this global log is checked. In [18], putting logs together is facilitated by assuming synchronized clocks, which is not a prerequisite to applying our analysis approach. Authors in [8] investigate the computational cost of log analysis w.r.t. graphs of MSCs. This cost is compared in different cases according to the quality of observations (local or tester observability i.e. whether one has a set of independent local logs or a globally ordered log) and the expressivity of the MSC graphs (presence of choice, loop or parallelism). The work echoes results for "MSC Membership" [1, 10] which state that this problem is NP-complete. The main factor of the cost blow-up lies in the fact that distributed actions can be equally re-ordered in multiple ways. Our work relates to such research, but we consider richer interactions (asynchronous communications, weak sequencing, no enforced fork-join, ...). As such, the language used in this paper is closer to the expressiveness of UML Sequence Diagrams. We therefore expect higher computational costs. Nevertheless, by applying a small-step semantics guided by the reading of the multi-trace, only pertinent parts of the search space are explored.

Plan. Sec.2 introduces our definition of multi-traces and recalls the concrete syntax of interactions as introduced in [16]. Sec.3 reformulates definitions from [16] so as to introduce the rewriting of interaction terms to define their semantics. We also extend [16] by defining a small-step semantics in the form of accepted multi-traces. Sec.4 is the core of our contribution. It presents our multi-trace analysis and some theoretical properties (termination, characterization of membership, NP-hardness). Sec.5 briefly introduces our prototype implementation HIBOU and Sec.6 concludes the paper.

2 MULTI-TRACES AND INTERACTIONS

Describing a DS requires distinguishing between its distinct independent sub-systems and the different messages those sub-systems can exchange. In this paper, those sub-systems are abstracted as so-called *lifelines* (classical terminology of interaction-based languages). We will denote L the set of all lifelines and M the set of all messages. In the rest of the paper, L and M will be left implicit.

The building blocks of both multi-traces and interactions are actions. An action is either the emission or the reception of a message m from or towards a lifeline l, denoted respectively $l!m$ and $l?m$. We denote $Act = \{l\Delta m \mid l \in L,\ \Delta \in \{!, ?\},\ m \in M\}$ the set of all actions. $Act(l)$ is the set of actions of the form $l\Delta m$. For an action act of the form $l!m$ or $l?m$, $lf(act)$ will stand for the lifeline l.

2.1 (Multi-)Traces

A trace characterizes an execution of a DS as a sequence[1] of actions, appearing in the order in which they occurred globally. From now on, the set L of lifelines will be assumed to be finite, and provided with a total order. By convention, when L is defined as the set $\{l_1, \cdots, l_n\}$, the indexes from 1 to n will be used to order the elements of L, i.e. $l_i < l_j$ iff $i < j$ in $\mathbb{N}$. Given $L = \{l_1, \cdots, l_n\}$, a multi-trace is a tuple of traces $\mu = (\sigma_1, \cdots, \sigma_n)$ where, for any $j \in [1, n]$, $\sigma_j \in Act(l_j)^*$ and where traces are ordered in the tuple respecting the order associated with the lifelines. A multi-trace describes the execution of a DS as the collection of traces locally observed on each sub-system. Multi-traces do not constrain orderings between actions occurring on different lifelines. $Mult = \prod_{l \in L} Act(l)^*$ is[2] the set of multi-traces. Def.2.1 introduces the projection operator $proj$ that projects any trace $\varsigma \in Act^*$ into a multi-trace $proj(\varsigma) \in Mult$.

Definition 2.1 (Trace Projection). $proj : Act^* \rightarrow Mult$ is s.t.:

- $proj(\epsilon) = (\epsilon, \cdots, \epsilon)$
- given $j \in [1, n]$ and $act \in Act(l_j)$ and $\varsigma \in Act^*$
  if $proj(\varsigma) = (\sigma_1, \cdots, \sigma_j, \cdots, \sigma_n)$ then
  $proj(act.\varsigma) = (\sigma_1, \cdots, act.\sigma_j, \cdots, \sigma_n)$.

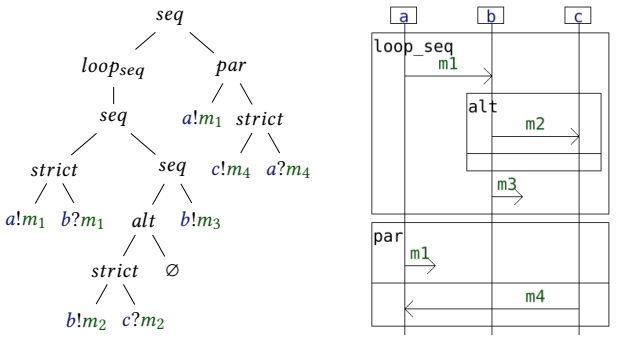
2.2 Interaction Language

Interactions synthesize possible executions of DSs by exhibiting the actions that can be observed and the possible orderings between them. As shown in Fig. 1 (left), interactions are binary trees whose leaves are actions. Precedence relations between 2 actions at different leaf positions are determined by the operators found in the inner nodes of the tree that separates those 2 positions.

Definition 2.2 (Interactions). The set Int of *interactions* is s.t.:

- $\emptyset \in Int$ and $Act \subset Int$,

[1] For a set X, X^* denotes the set of sequences of elements of X with ϵ being the empty sequence and the dot notation (.) being the concatenation operator.
[2] $\prod_{l \in L} A_l$ denotes the Cartesian product of sets A_{l_i}, i.e. the set of tuples $(a_1, \ldots, a_n)$ with $a_i \in A_{l_i}$ and with the indexes l_i ordered accordingly to the ordered set L.



interaction term diagram view

Figure 1: Example interaction

- for $(i_1, i_2) \in Int^2$ and $f \in \{strict, seq, alt, par\}$, $f(i_1, i_2) \in Int$,
- for $i \in Int$ and $f \in \{strict, seq, par\}$, $loop_f(i) \in Int$.

The empty interaction $\emptyset$ and any action of Act are basic interactions. $seq(i_1, i_2)$ (weak sequencing) indicates that actions specified by i_1 must occur before those of i_2 iff they occur on the same lifeline. In contrast, $strict(i_1, i_2)$ (strict sequencing) imposes that actions specified by i_1 must occur before those of i_2 in any case. $par(i_1, i_2)$ allows actions from i_1 and i_2 to be fully interleaved while $alt(i_1, i_2)$ (exclusive alternative) specifies that either actions specified by i_1 or by i_2 occur. As for the loop operators, $loop_f$ with $f \in \{seq, strict, par\}$, the index f indicates with which binary operator loop unrollings have to be composed: in other words $loop_f(i_1)$ is equivalent to the term $alt(\emptyset, f(i_1, loop_f(i_1))$ (here we detailed the choice between not unrolling ($\emptyset$) and unrolling once).

Interactions can be illustrated by diagrams (cf. right part of Fig. 1). Lifelines are depicted as vertical lines and actions $l\Delta m$ as arrows carrying their specific message m and originating from or pointing towards their specific lifeline l. The passing of a message from a lifeline to another is modelled using the *strict* operator (e.g. $strict(a!m, b?m)$ to denote the passing of m from a to b). A message passing is depicted as an arrow from source to target lifeline.

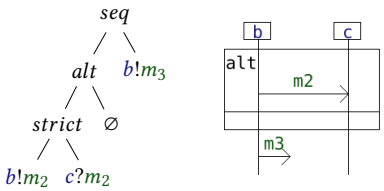


Figure 2: Small example

Let us consider the example from Fig. 2 (subterm of the one from Fig. 1). Firstly, b can either send m_2 to c or not send anything. This choice is modelled by the *alt* alternative operator. Secondly, b must send m_3 to the environment. The implicit sequencing that we have described in natural language with the adverbs "firstly" and "secondly" is modelled by the *seq* weak sequencing operator, which, unlike the other operators that are drawn explicitly with boxes, is implicitly represented by the top to bottom direction.

The semantics of an interaction i is defined as a set of global traces $Accept(i)$ or of multi-traces $AccMult(i)$. Fig. 3 enumerates those semantics for the interaction from Fig. 2. An interleaving between $b!m_3$ and $c?m_2$ is noticeable in $Accept(i)$ but not in $AccMult(i)$.

$$Accept(i) = \left\{ \begin{array}{l} b!m_2.c?m_2.b!m_3, \\ b!m_2.b!m_3.c?m_2, \\ b!m_3 \end{array} \right\}$$

$$AccMult(i) = \left\{ \begin{array}{l} (b!m_2.b!m_3,\ c?m_2), \\ (b!m_3,\ \epsilon) \end{array} \right\}$$

Figure 3: Semantics of example from Fig. 2

3 ACCEPTED (MULTI-)TRACES

To formally define the set of accepted (multi-)traces of an interaction i, we reformulate semantic rules from [16] without relying on some denotational counterpart (in particular, without using precedence relations between actions, as in [12, 16]). To do this, in Sec.3.1, we extract information from the term structure of interactions. This information is used to define, in Sec.3.2, the small-step interaction execution function χ grounding the operational approach. Finally, in Sec.3.3, we give two interaction semantics: *Accept*, based on global traces, and *AccMult*, obtained by projection of *Accept*.

3.1 Static analysis of interactions

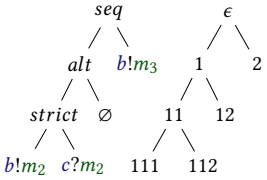


Figure 4: Positions

As an interaction i can contain several occurrences of the same action act, small-steps do not correspond to transformations of the form $i \xrightarrow{act} i'$ but rather $i \xrightarrow{act@p} i'$ where p indicates the position of a specific occurrence of act within i. To do so, we use positions expressed in the Dewey Decimal Notation [9]. As the arity of operators is at most 2, positions are defined as elements of $\{1, 2\}^*$. A sub-interaction of an interaction i at position p is denoted $i_{|p}$. Fig. 4 illustrates positions within the interaction from Fig. 2. For example, the subterm $strict(b!m_2, c?m_2)$ is at position 11 of the whole interaction term $seq(alt(strict(b!m_2, c?m_2), \emptyset), b!m_3)$. Moreover, for any set $P \in \mathcal{P}(\{1, 2\}^*)$ and $x \in \{1, 2\}$, $x.P$ stands for the set $\{x.p \mid p \in P\}$.

As a prerequisite to defining steps of the form $i \xrightarrow{act@p} i'$, we need to introduce several intermediate functions on interaction terms.

Function exp_ϵ defined in Def.3.1 assesses statically whether or not an interaction accepts/expresses the empty trace ϵ. Naturally $\emptyset$ only accepts ϵ, while interactions $act \in Act$ do not (act must be executed). Similarly, any loop accepts ϵ because it is possible to repeat 0 times its content. The treatment of binary operators differs according to their intuitive meaning: for *alt*, it is sufficient that one of the two direct sub-interactions accepts ϵ, while for the scheduling operators (*seq*, *strict* and *par*), both have to accept ϵ.

Function *avoids* defined in Def.3.1 states, for an interaction i and a lifeline l, whether or not i accepts an execution that involves no actions occurring on l. The empty interaction $\emptyset$ avoids every lifeline since the only trace specified by $\emptyset$ is the empty trace. An action $l'\Delta m$ avoids the lifeline l iff it occurs on a different lifeline, i.e. $l' \neq l$. Then, as for exp_ϵ, *avoids* is defined inductively accordingly the top operator of the interaction term. Apart from the basic cases ($\emptyset$ and $l\Delta m$), the two functions are quite similar in their description.

Definition 3.1 (Emptiness & Avoiding). We define the functions $exp_\epsilon : Int \to bool$ and $avoids : Int \times L \to bool$ s.t. for any $l \in L$:

- $exp_\epsilon(\emptyset) = \top$ and $avoids(\emptyset, l) = \top$
- for $act = l'\Delta m \in Act$, $exp_\epsilon(act) = \bot$ and $avoids(act, l) = (l' \neq l)$,
- for $i = f(i_1, i_2)$ with $f \in \{strict, seq, par\}$, $exp_\epsilon(i) = exp_\epsilon(i_1) \wedge exp_\epsilon(i_2)$ and $avoids(i, l) = avoids(i_1, l) \wedge avoids(i_2, l)$
- for $i = alt(i_1, i_2)$, $exp_\epsilon(i) = exp_\epsilon(i_1) \vee exp_\epsilon(i_2)$ and $avoids(i, l) = avoids(i_1, l) \vee avoids(i_2, l)$,
- for $i = loop_f(i')$ with $f \in \{strict, seq, par\}$, $exp_\epsilon(i) = \top$ and $avoids(i, l) = \top$.

Among all action leaves of i, only some are immediately executable, in the sense that they can be the first element of at least one trace of i. Function $front$ (for *frontier*) in Def.3.2, determines the positions of all such actions. The selection of those actions depend on which operators are encountered within the tree structure of the interaction. Sequencing operators *strict* and *seq*, by enforcing precedence relations, may prevent some actions (from the sub-interaction on the right of the term) from being in the frontier.

Definition 3.2 (Frontier). $front : Int \rightarrow \mathcal{P}(\{1,2\}^*)$ is the function s.t.:

- $front(\emptyset) = \emptyset$ and for all $act \in Act$, $front(act) = \{\epsilon\}$,
- for all $(i_1, i_2) \in Int^2$:
  - $front(strict(i_1, i_2)) = \begin{cases} 1.front(i_1) \cup 2.front(i_2) & \text{if } exp_\epsilon(i_1) = \top \\ 1.front(i_1) & \text{else} \end{cases}$
  - $front(seq(i_1, i_2)) = 1.front(i_1) \cup \{p \mid p \in 2.front(i_2)$ and $avoids(i_1, lf(i_{|p}))\}$,
  - $front(f(i_1, i_2)) = 1.front(i_1) \cup 2.front(i_2)$ for $f \in \{alt, par\}$
  - $front(loop_f(i_1)) = 1.front(i_1)$ for $f \in \{strict, seq, par\}$.

For any $p \in front(i)$, $i_{|p}$ is a called a frontier action.

The empty interaction has an empty frontier: $front(\emptyset) = \emptyset$. For any action act, $front(act) = \{\epsilon\}$ (ϵ is the position of act which is immediately executable). For i of the form $f(i_1, i_2)$, $front(i)$ is inferred from $front(i_1)$ and $front(i_2)$ in accordance with the intuitive sense of f operator f. In all cases, actions occurring at positions in $front(i_1)$ are immediately executable in i. Indeed, the term being read from left to right, all operators, if they introduce ordering constraints, will only do so on the right sub-interaction i_2. Thus $1.front(i_1)$ is included in $front(i)$. If $f = alt$ or $f = par$, $2.front(i_2)$ is also included in $front(i)$ because no constraint may prevent the execution of actions from i_2. If $f = strict$, any action from i_2 can only be executed if no action from i_1 is (otherwise it would violate the strict sequencing). Thus $2.front(i_2)$ is included in $front(i)$ iff i_1 accepts the empty trace. If $f = seq$, elements p from $2.front(i_2)$ are included in $front(i)$ iff i_1 accepts an execution that does not involve the lifeline on which the action $i_{|p}$ occurs.

Fig. 5 illustrates the definition of $front$ on the example from Fig. 1. Given the 8 different actions on leaves, we have $front(i) \subseteq \{1111, 1112, 112111, 112112, 1122, 21, 221, 222\}$. Actions on the right of every *strict* operators are prevented from being executed by those on their left and as such are not in the frontier. This eliminates $\{1112, 112112, 222\}$. $b!m_2$ and $b!m_3$ are prevented from being executed by $b?m_1$ which is a cousin on their left w.r.t the *seq* operator at position 11. This eliminates $\{1112, 1122\}$. Then, by elimination, $front(i) = \{1111, 21, 221\}$ so that the 3 actions placed at

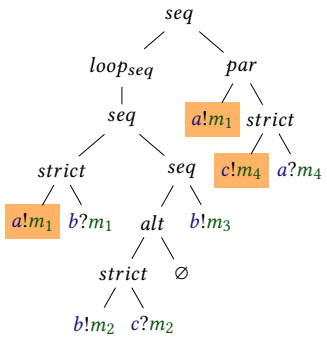


Figure 5: Frontier actions (highlighted)

these positions are the only actions that can start a trace accepted by interaction i.

3.2 Interaction Execution

Small-steps of the operational semantics consist in transforming an interaction i having position p in its frontier into an interaction i' s.t. i' characterizes in intentions all the possible futures of the execution of the action $i_{|p}$ according to i.

We define a function *prune*, that associates to any interaction i that may avoid l (i.e. $avoids(i, l) = \top$), a new interaction, which characterizes exactly all the executions of i that do not involve lifeline l. In other words, $prune(i, l)$ computes an interaction whose accepted traces are exactly those of i that have no actions occurring on lifeline l.

Definition 3.3 (Pruning). The function $prune : Int \times L \rightarrow Int$ is defined for couples (i, l) in $Int \times L$ verifying $avoids(i, l)$ by:

- $prune(\emptyset, l) = \emptyset$ and for any $act \in Act$, $prune(act, l) = act$
- for any $(i_1, i_2) \in Int^2$, $prune(alt(i_1, i_2), l)$ is:
  - $prune(i_1, l)$ if $avoids(i_1, l) \wedge \neg avoids(i_2, l)$
  - $prune(i_2, l)$ if $\neg avoids(i_1, l) \wedge avoids(i_2, l)$
  - $alt(prune(i_1, l), prune(i_2, l))$ if $avoids(i_1, l) \wedge avoids(i_2, l)$
- for any $(i_1, i_2) \in Int^2$ and any $f \in \{strict, seq, par\}$, we have $prune(f(i_1, i_2), l) = f(prune(i_1, l), prune(i_2, l))$
- for any $i \in Int$ and any $f \in \{strict, seq, par\}$:
  - $prune(loop_f(i), l) = loop_f(prune(i, l))$ if $avoids(i, l)$
  - $prune(loop_f(i), l) = \emptyset$ if $\neg avoids(i, l)$

The use of *prune* is illustrated on the sub-interaction $i_{|1}$ highlighted in blue on Fig.6. $i_{|1}$ is such that $avoids(i_{|1}, c)$ is true, hence $prune(i_{|1}, c)$ can be applied. The blue lines represent the rewriting orchestrated in $prune(i_{|1}, c)$. We have to eliminate $c?m_2$ the only action occurring in $i_{|1}$ on c. As its parent is a scheduling operator (*strict*), it must also be eliminated. The grand-parent node is an *alt* operator. The right cousin underneath this *alt* is $\emptyset$, which "avoids" c. Thus, the choice of the right branch of this *alt* can be forced to solve the pruning. The remaining interaction $prune(i_{|1}, c)$ is $loop_{seq}(seq(strict(a!m_1, b?m_1), b!m_3))$ (where we simplified $seq(\emptyset, b!m_3)$ into $b!m_3$). It does not contain any action occurring on c.

The next definition introduces the "eχecution" function χ. For an interaction i and a position p in $front(i)$, $\chi(i, p)$ returns a pair $(i', i_{|p})$. Here, interaction i' is such that all traces accepted by i and which start by $i_{|p}$ (the action specifically at position p, and not

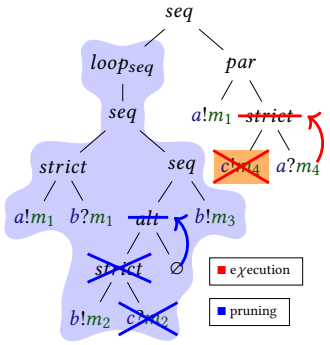


Figure 6: Illustration of a small-step

another identical action somewhere else in i) i.e. of the form $i_{|p}.\varsigma$ are such that ς is accepted by i'; and, reciprocally, for any ς accepted by i' then $i_{|p}.\varsigma$ is accepted by i. i' is therefore the "continuation" of i after the occurrence of $i_{|p}$. $\chi(i,p)$ is defined by induction on the term structure of i, and by case depending on whether position p starts by 1 or 2, (i.e. whether p belongs to the left or right subterm of i) or is $p = \epsilon$ (then i is the action to execute).

Definition 3.4 (Interaction Execution). The function $\chi : Int \times \{1,2\}^* \rightarrow Int \times Act$, defined for pairs (i,p) verifying $p \in front(i)$ is s.t.:

- for any $act \in Act$, $\chi(act, \epsilon) = (\varnothing, act)$
- for any $(i_1, i_2) \in Int^2$, $f \in \{strict, seq, par\}$ and $p_1 \in front(i_1)$, let us denote $\chi(i_1, p_1) = (i'_1, act)$, then:
  - $\chi(alt(i_1, i_2), 1.p_1) = (i'_1, act)$,
  - $\chi(f(i_1, i_2), 1.p_1) = (f(i'_1, i_2), act)$,
  - $\chi(loop_f(i_1), 1.p_1) = (f(i'_1, loop_f(i_1)), act)$,
- for any $(i_1, i_2) \in Int^2$ and $p_2 \in front(i_2)$, let us denote $\chi(i_2, p_2) = (i'_2, act)$, then:
  - $\chi(alt(i_1, i_2), 2.p_2) = (i'_2, act)$,
  - $\chi(strict(i_1, i_2), 2.p_2) = (i'_2, act)$,
  - $\chi(seq(i_1, i_2), 2.p_2) = (seq(prune(i_1, lf(act)), i'_2), act)$,
  - $\chi(par(i_1, i_2), 2.p_2) = (par(i_1, i'_2), act)$.

χ is defined on the cases authorized by its precondition $p \in front(i)$. If $i \in Act$, p can only be ϵ (and vice-versa). In this case $\chi(i, \epsilon) = (\varnothing, i)$ since the action i is executed and nothing remains to be executed. In any other case, p is either of the form $1.p_1$ or $2.p_2$, meaning that the action to be executed is resp. in the left or right sub-interaction. Then the result of $\chi(i,p)$ is a reconstruction of the interaction term from resp. the result of $\chi(i_1, p_1)$ and i_2 or the result of $\chi(i_2, p_2)$ and i_1. The most subtle case occurs when $p = 2.p_2$ and $i = seq(i_1, i_2)$. The precondition $p \in front(i)$ implies that $i_{|p} \in Act$ and that the left child i_1 avoids $lf(i_{|p})$. In this case, to construct $\chi(i, 2.p_2)$, χ does not use i_1 but rather $prune(i_1, lf(i_{|p}))$, where all traces involving $lf(i_{|p})$ have been eliminated while preserving all others. Fig. 6 depicts the execution process applied to $c!m_4$ (at position 221) in the interaction i of Fig. 1: $\chi(i, 221)$ is $(seq(i', par(a!m_1, a?m_4)), c!m_4)$ with $i' = prune(i_{|1}, c)$, previously computed as $loop_{seq}(seq(strict(a!m_1, b?m_1), b!m_3))$. The computation of $\chi(i, 221)$ is also visualized as the first step of the right branch of Fig. 7.

To conclude, the χ function fulfills our objectives of defining the semantics of interactions using elementary steps of the form $i \xrightarrow{act@p} i'$, which is a readable and graphic reformulation of the equality $\chi(i,p) = (i', act)$.

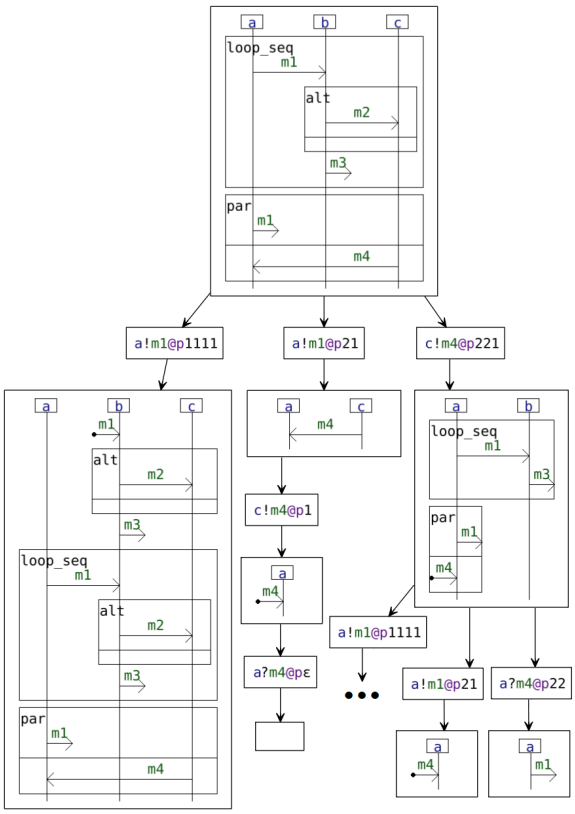


Figure 7: Execution tree illustration

3.3 Definition of accepted (multi-)traces

The small-step approach to compute semantics consists in exploring an execution tree representing all possible successions of transformations $i \xrightarrow{act@p} i'$, starting from an initial interaction i_0. An accepted trace then corresponds to a sequence $act_1.\cdots.act_n$ obtained from a path $i_0 \xrightarrow{act_1@p_1} i_1 \cdots \xrightarrow{act_n@p_n} i_n$ with i_n a terminal interaction, i.e. accepting ϵ. By grouping all such paths together, we obtain a tree whose nodes are interactions and arcs are labelled by couples (p, act) denoted $act@p$. For a node i, child nodes are interactions i' obtained via the execution of any frontier action $act = i_{|p}$ with $p \in front(i)$. Any such child node i' corresponds to an interaction accepting traces that are suffixes of traces accepted by i and which start with act. Fig. 7 illustrates this process on the interaction from Fig.1 (a partially drawn execution tree is represented). As already mentioned, this interaction has three frontier actions (immediately executable), at positions 1111, 21 and 221 respectively, which gives rise to 3 direct successor nodes. The path leading to the empty interaction (white square □) yields the trace $a!m_1.c!m_4.a?m_4$.

Let us note that, when an interaction contains a loop operator, sequences of consecutive executions of actions can be arbitrarily long (as suggested by the use of • • • in Fig.7). Indeed, a given execution can involve an arbitrarily large number of loop unfoldings, depending on whether small-steps occurring during the execution are obtained by unfolding, ignoring or pruning-out a loop. In a given small step $i \xrightarrow{act@p} i'$, if action act at position p is within a

loop operator in i, then this loop is unfolded in i'. If not, then this loop might have been ignored (left unchanged) or might have been eliminated (pruned) in i'. In this aspect, our treatment of loops is similar to the small-step unfolding of the Binary Kleene Star operator introduced in [6] (in a process algebra). We do not need to explicit the fact of not taking a loop with some form of a "skip" action and a semantical rule as in [21]. If an action that is supposed to occur "after" a certain loop is executed then the loop is eliminated (or its content pruned) during the rewriting of the interaction term by a call of the *prune* function. Loops making possible arbitrarily long executions, we consider in Def.3.5 semantics consisting of sets of arbitrary long (multi-)traces.

Definition 3.5 (Semantics). $Accept : Int \rightarrow \mathcal{P}(Act^*)$ and $AccMult : Int \rightarrow \mathcal{P}(Mult)$ are s.t. for any $i \in Int$:

$$Accept(i) = empty(i) \cup \left\{ act.Accept(i') \;\middle|\; \begin{array}{l} \exists\, p \in front(i), \\ \chi(i,p) = (i', act) \end{array} \right\}$$

$$AccMult(i) = \{proj(\varsigma) \mid \varsigma \in Accept(i)\}$$

with: $empty(i) = \{\epsilon\}$ if $exp_\epsilon(i)$ and $empty(i) = \emptyset$ otherwise.

4 MULTI-TRACE MEMBERSHIP ANALYSIS

4.1 Principle

We define a process able to decide whether or not a multi-trace μ is accepted by an interaction i. Its key principle is to construct traces accepted by i that project on μ. Constructing those traces is based on elementary steps $(i,\mu) \rightsquigarrow (i',\mu')$ s.t. $\chi(i,p) = (i', act_j)$ for some $p \in front(i)$ with $act_j \in Act(l_j)$, $\mu = (\sigma_1, \cdots, act_j.\sigma_j, \cdots, \sigma_n)$ and $\mu' = (\sigma_1, \cdots, \sigma_j, \cdots, \sigma_n)$. By considering all possible $p \in front(i)$, and by iterating those steps of computation, the process builds a tree whose paths are of the form $(i_0,\mu_0) \rightsquigarrow \cdots \rightsquigarrow (i_p,\mu_p) \cdots \rightsquigarrow (i_q,\mu_q)$, denoted as $(i_0,\mu_0) \overset{*}{\rightsquigarrow} (i_q,\mu_q)$.

At each step $(i,\mu) \rightsquigarrow (i',\mu')$, the size of the multi-trace decreases by one. Hence, any path eventually reaches a point where it is no longer possible to find a next step. This halting of the process can occur in 2 cases.

(1) Either the process reaches a state (i_q,μ_q) where μ_q is not empty and no frontier action of i_q matches some first elements in μ_q. In that case the sequence of actions that leads to (i_q,μ_q) is not a trace accepted by i and a local verdict $UnCov$ (for "multi-trace not covered") is associated to $(i_0,\mu_0) \overset{*}{\rightsquigarrow} (i_q,\mu_q)$.

(2) Or the process reaches a state $(i_q,(\epsilon,\cdots,\epsilon))$. Here, all actions of μ have been consumed to form a given global trace ς. The process then checks if ς is accepted by i (which happens iff i_q accepts the empty trace). If the answer is yes then $(i_0,\mu_0) \overset{*}{\rightsquigarrow} (i_q,(\varepsilon,\cdots,\varepsilon))$ is associated with a coverage verdict Cov (for "multi-trace covered"). Otherwise, the verdict $UnCov$ is associated to the path.

If there exists a path leading to Cov, the global verdict is $Pass$. If no such path exists, the global verdict is $Fail$.

4.2 Definition of analysis process

Multi-trace analysis relies on 4 rules, denoted $R1$, $R2$, $R3$ and $R4$ and given in Def.4.1. Those rules define a directed graph $\mathbb{G}$ in which vertices are either a tuple $(i,\mu) \in Int \times Mult$ or a coverage verdict $v \in \{Cov, UnCov\}$. We note $\mathbb{V} = \{Cov, UnCov\} \cup (Int \times Mult)$ the set of vertices. For x in $\{1,2,3,4\}$, the rule $(Rx)\frac{v}{v'}\ cond$, with $v \in Int \times Mult$ and $v' \in \mathbb{V}$ specifies edges of the form $v \rightsquigarrow v'$ of that graph, provided that v satisfies condition $cond$.

Definition 4.1 (Rules of Multi-Trace Analysis). The analysis relation $\rightsquigarrow \subseteq \mathbb{V} \times \mathbb{V}$ is defined as:

$$\text{(R1)}\ \frac{i \qquad (\epsilon,\cdots,\epsilon)}{Cov}\ exp_\epsilon(i)$$

$$\text{(R2)}\ \frac{i \qquad (\epsilon,\cdots,\epsilon)}{UnCov}\ \neg exp_\epsilon(i)$$

$$\text{(R3)}\ \frac{i \qquad (\sigma_1,\cdots,act.\sigma_k,\cdots,\sigma_n)}{i' \qquad (\sigma_1,\cdots,\sigma_k,\cdots,\sigma_n)}\ \begin{array}{l} \exists\, p \in front(i) \text{ s.t.} \\ \chi(i,p) = (i', act) \end{array}$$

$$\text{(R4)}\ \frac{i \qquad (\sigma_1,\cdots,\sigma_n)}{UnCov}\ \left\{ \begin{array}{l} (\sigma_1,\cdots,\sigma_n) \neq (\epsilon,\cdots,\epsilon) \\ \wedge \left(\begin{array}{l} \forall\, j \in [1,n],\ \forall\, p \in front(i), \\ (\sigma_j \neq \epsilon) \Rightarrow (fst(\sigma_j) \neq i_{|p}) \end{array} \right) \end{array} \right.$$

where $fst(\sigma)$ denotes the first element of a non empty sequence σ.

Let us comment Def.4.1. Vertices of the form (i,μ) are not sinks. If μ is the empty multi-trace, given that $exp_\epsilon(i)$ can either be $true$ or $false$, either $R1$ or $R2$ applies and so there exists an outgoing edge from any $(i,(\epsilon,\ldots,\epsilon))$. If $\mu \neq (\epsilon,\ldots,\epsilon)$, one can either have or not have matches between frontier actions and multi-trace component heads. Hence, an outgoing edge exists accordingly to $R3$ or $R4$. So, coverage verdicts $\{Cov, UnCov\}$ are the 2 only sinks of $\mathbb{G}$.

Rules $R1$, $R2$ and $R4$ specify edges from vertices of the form (i,μ) to coverage verdicts. The rule $R3$ specifies edges $(i,\mu) \rightsquigarrow (i',\mu')$ such that (1) there exists an action act occurring in i at position $p \in front(i)$ matching a head action act_j of μ, i.e. $\mu = (\sigma_1,\cdots,act_j.\sigma'_j,\cdots,\sigma_n)$, (2) i' is defined by $\chi(i,p) = (i', act_j)$, and (3) μ' is the multi-trace μ in which we have removed act_j, i.e. $\mu' = (\sigma_1,\cdots,\sigma'_j,\cdots,\sigma_n)$. Let us note that for a vertex (i,μ), there are at most $|front(i)|$ possible applications of the rule $R3$ with $|front(i)|$ bounded by the number of occurrences of actions in i.

Let us consider $|\mu|$ the number of actions occurring in a multi-trace μ, i.e. the sum of lengths of its component traces. Let us extend this notation to vertices, that is, $|(i,\mu)|$ defined as $|\mu|$, and $|Cov|$ and $|UnCov|$ defined as -1. For any edge $v \rightsquigarrow v'$ of $\mathbb{G}$, we have $|v'| < |v|$ with $|v'| \geq -1$. Consequently, the successive application of the rules strictly decrements the size of nodes and from any vertex (i,μ), any maximal outgoing path is finite, and terminates in a coverage verdict in $\{Cov, UnCov\}$ (since (i,μ) are not sinks of $\mathbb{G}$). Thus, $\mathbb{G}$ is an acyclic graph. With the notation $v \overset{*}{\rightsquigarrow} v'$ to indicate that there is a path from v to v' in $\mathbb{G}$, we define multi-trace analysis.

Definition 4.2 (Multi-Trace Analysis). We define $\omega : Int \times Mult \rightarrow \{Pass, Fail\}$ such that for any $i \in Int$ and $\mu \in Mult$ we have:

- $\omega(i,\mu) = Pass$ iff there exists a path $(i,\mu) \overset{*}{\rightsquigarrow} Cov$
- $\omega(i,\mu) = Fail$ otherwise; i.e. for all path $(i,\mu) \overset{*}{\rightsquigarrow} v$ with $v \in \{Cov, UnCov\}$, then $v = UnCov$

The function ω is well-defined. Indeed, we established that all maximal paths from a vertex (i_0,μ_0) have a maximum length of $|\mu| + 1$ and end on a coverage verdict (Cov or $UnCov$). As besides, each intermediate vertex (i,μ) between (i_0,μ_0) and a coverage verdict has a number of children bounded by the number of actions of i, then the set of vertices reachable from (i_0,μ_0) is finite.

4.3 Correctness w.r.t the semantics

We now prove that the function ω in charge of analysing multi-traces w.r.t. an interaction captures exactly its semantics defined

by the step-by-step execution function χ given in Sec.3. More precisely, we will prove that for any (i,μ) in $Int \times Mult$, $\omega(i,\mu) = Pass$ iff $\mu \in AccMult(i)$ (and by extension, $\omega(i,\mu) = Fail$ iff $\mu \notin AccMult(i)$). Given that $AccMult(i)$ is the set of projected global traces of $Accept(i)$, it then suffices to prove that for any trace $\varsigma \in Act^*$ we have $\omega(i, proj(\varsigma)) = Pass$ iff $\varsigma \in Accept(i)$. Below, Th.4.3 and Th.4.4 resp. correspond to the $\Leftarrow$ and $\Rightarrow$ implication of this "*iff*".

Theorem 4.3 (*Accept* implies *Pass*). *For any* $(i,\varsigma) \in Int \times Act^*$:

$$(\varsigma \in Accept(i)) \Rightarrow (\omega(i, proj(\varsigma)) = Pass)$$

Proof. Let us reason by induction on the trace ς.
• $\varsigma = \epsilon$. Let us consider an interaction i s.t. $\epsilon \in Accept(i)$. We have $proj(\epsilon) = (\epsilon, \cdots, \epsilon)$. As $\epsilon \in Accept(i)$, then $exp_\epsilon(i) = \top$ and R_1 is applicable from $(i, (\epsilon, \cdots, \epsilon))$. We obtain $\omega(i, (\epsilon, \cdots, \epsilon)) = Pass$.
• $\varsigma = act.\varsigma'$. Let us consider i s.t. $\varsigma \in Accept(i)$. The induction hypothesis on ς' is: "$\forall\, i' \in Int,\ (\varsigma' \in Accept(i')) \Rightarrow (\omega(i', proj(\varsigma')) = Pass)$". As $act.\varsigma' \in Accept(i)$, then there exists i' in Int and $p \in front(i)$ s.t. $\chi(i,p) = (i', act)$ and $\varsigma' \in Accept(i')$. Let us consider the index j such that $proj(act.\varsigma') = (\sigma_1, \cdots, act.\sigma_j, \cdots, \sigma_n)$. Given that $\chi(i,p) = (i', act)$, $R3$ can be applied so that, $(i, (\sigma_1, \cdots, act.\sigma_j, \cdots, \sigma_n)) \rightsquigarrow (i', (\sigma_1, \cdots, \sigma_j, \cdots, \sigma_n))$ with $(\sigma_1, \cdots, \sigma_j, \cdots, \sigma_n) = proj(\varsigma')$. We have $(\omega(i', proj(\varsigma')) = Pass)$ by induction, i.e. there exists a path $(i', proj(\varsigma')) \overset{*}{\rightsquigarrow} Cov$. By preceding this path with $(i, proj(act.\varsigma')) \rightsquigarrow (i', proj(\varsigma'))$, we get $(i, (\sigma_1, \cdots, act.\sigma_j, \cdots, \sigma_n)) \overset{*}{\rightsquigarrow} Cov$ and $\omega(i, proj(\varsigma)) = Pass$. □

Theorem 4.4 (*Pass* implies *Accept*). *For any* $(i,\mu) \in Int \times Mult$:

$$(\omega(i,\mu) = Pass) \Rightarrow \left(\exists\, \varsigma \in Act^*\ s.t.\ proj(\varsigma) = \mu\ and\ \varsigma \in Accept(i)\right)$$

Proof. Let us reason by induction on the size of μ, i.e. on $|\mu|$.
• $|\mu| = 0$. Let us consider i s.t. $\omega(i,\mu) = Pass$. By $|\mu| = 0$, $\mu = (\epsilon, \cdots, \epsilon)$. Since $\omega(i, (\epsilon, \cdots, \epsilon)) = Pass$, $R1$ must apply and this implies that $exp_\epsilon(i) = \top$ and consequently $\epsilon \in Accept(i)$. Therefore the property holds at length 0.
• $|\mu| = z + 1$ with $z \geq 0$. Let us consider i s.t. $\omega(i,\mu) = Pass$. The induction hypothesis states that "for all $(i',\mu') \in Int \times Mult$ with $|\mu'| = z$, $(\omega(i',\mu') = Pass) \Rightarrow (\exists\, \varsigma' \in Act^*$ s.t. $proj(\varsigma') = \mu'$ and $\varsigma' \in Accept(i'))$". Since $\omega(i,\mu) = Pass$, there exists a path $(i,\mu) \overset{*}{\rightsquigarrow} Cov$. As noticed in Sec. 4.2, each edge of a maximal path exactly consumes one action, with the exception of the last edge leading to the coverage verdict. Thus the path starts with an edge of form $(i,\mu) \rightsquigarrow (i',\mu')$ with $|\mu'| = z$ and we have then $(i',\mu') \overset{*}{\rightsquigarrow} Cov$. By definition, $\omega(i',\mu') = Pass$. By induction, there exists a trace ς' s.t. $proj(\varsigma') = \mu'$ and $\varsigma' \in Accept(i')$. $(i,\mu) \rightsquigarrow (i',\mu')$ corresponds to the consumption of an action act which matches a frontier action $i_{|p}$ of i. By definition, the trace $\varsigma = act.\varsigma'$ verifies $proj(\varsigma) = \mu$ and $\varsigma \in Accept(i)$. □

The two theorems demonstrate that $\omega(i,\mu) = Pass$ characterizes the membership of a multi-trace μ to $AccMult(i)$. A Coq proof, formalizing our approach and which includes the 2 previous demonstrations is available online[4].

The computational cost of ω varies greatly depending on the initial (i,μ) couple. We demonstrate the NP-hardness of this membership problem through a reduction of the 1-in-3-SAT problem [22] that is inspired by [1, 8, 10] but requires the construction of a different reduction of which we discuss in the following.

4.4 Discussion on Complexity

1-in-3-SAT [22] is a particular Boolean satisifiability problem. Let us consider a set of $p \geq 1$ boolean variables $V = \{v_1, \cdots, v_p\}$ and a set of $q \geq 1$ clauses $\{C_1, \cdots, C_q\}$ in 3-CNF form i.e. s.t. for any $j \in [1,q]$, $C_j = \alpha_j \vee \beta_j \vee \gamma_j$ with $\alpha_j, \beta_j, \gamma_j$ in $V \cup \overline{V}$, $^-$ being the usual negation operator. The 1-in-3-SAT problem on formula $\phi = C_1 \wedge \cdots \wedge C_q$ then consists in finding $\rho : V \to \{\top, \bot\}$ s.t.[3] $\rho \models \phi$ and s.t. for any clause C_j, only one in the three literals α_j, β_j, or γ_j is set to $\top$. In the following, we sketch a reduction proof which states that any 1-in-3-SAT problem can be reduced to the multi-trace membership problem for a given $(i,\mu) \in Int \times Mult$ (i.e. whether or not $\mu \in AccMult(i)$).

Let us consider the reduction of 1-in-3-SAT in the simple case where $p = 4$ and $q = 2$. This approach can then be extended to include any other case.

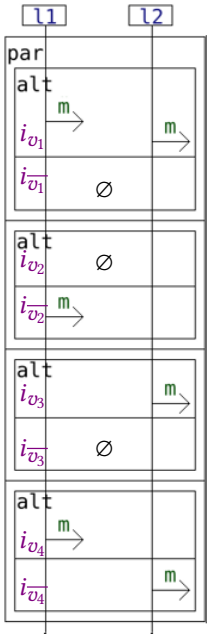


Figure 8

From formula $\phi = C_1 \wedge C_2$, we define an interaction i via a 1-on-1 transformation. This i is of the form exemplified on Fig.8 i.e. a parallelization of 4 alternatives $alt(i_v, i_{\overline{v}})$ s.t. for any $x \in V \cup \overline{V}$, i_x is s.t. if x occurs:
• in C_1 and C_2 then $i_x = seq(l_1!m, l_2!m)$
• in C_1 but not in C_2 then $i_x = l_1!m$
• in C_2 but not in C_1 then $i_x = l_2!m$
• neither in C_1 nor in C_2 then $i_x = \varnothing$

For instance, with $C_1 = (v_1 \vee \overline{v_2} \vee v_4)$ and $C_2 = (v_1 \vee v_3 \vee \overline{v_4})$, Fig.8 gives the corresponding interaction.

This 1-in-3-SAT problem ϕ is equivalent to the multi-trace membership problem $\mu = (l_1!m, l_2!m) \in AccMult(i)$. Indeed, in a given execution of i, component $\sigma_1 = l_1!m$ of μ is expressed exactly once iff exactly one of the sub-interactions i_{α_1}, i_{β_1} or i_{γ_1} is "chosen" during the execution of i. Given that the parent interaction (within i) of sub-interaction i_{α_1} (same reasoning for i_{β_1} and i_{γ_1}) is of the form $alt(i_{\alpha_1}, i_{\overline{\alpha_1}})$ (or with the order of branches inverted), "chosen" means that the exclusive branch that hosts i_{α_1} is chosen over that which hosts $i_{\overline{\alpha_1}}$.

The expression of component σ_1 on lifeline l_1 is therefore equivalent to the satisfaction of clause C_1 in 1-in-3-SAT. In our example, with $C_1 = (v_1 \vee \overline{v_2} \vee v_4)$, the fact that $\rho \models C_1$ with $\rho : [v_1 \to \bot, v_2 \to \top, v_3 \to \top, v_4 \to \top]$ is equivalent to the fact that $l_1!m$ is expressed exactly once during the execution of i when $i_{\overline{v_1}}$ is chosen over i_{v_1}, i_{v_2} over $i_{\overline{v_2}}$, i_{v_3} over $i_{\overline{v_3}}$, and i_{v_4} over $i_{\overline{v_4}}$.

The same reasoning applies for the relationship between C_2 and $\sigma_2 = l_2!m$. In other words, during the execution of i, given the use of exclusive alternative operators in $alt(i_v, i_{\bar{v}})$ sub-terms, the choice of either one of the *alt* branch constitutes an assignment of Boolean variable v. The overall parallel composition then simulates

[3] "$\rho \models \phi$" is the usual satisfaction relation in propositional logic.

all possible variable assignments (i.e. the search space for ρ). Then, the satisfaction of ϕ as the conjunction of clauses C_1 and C_2 in 1-in-3-SAT is equivalent to that of $\mu = (\sigma_1, \sigma_2) \in AccMult(i)$. Indeed, the same ρ must be used to solve both C_1 and C_2 and the same global execution of i must be used to consume both σ_1 and σ_2 exactly.

In our example, $\phi = (v_1 \vee \overline{v_2} \vee v_4) \wedge (v_1 \vee v_3 \vee \overline{v_4})$ is solvable in 1-in-3-SAT by $\rho : [v_1 \rightarrow \bot, v_2 \rightarrow \top, v_3 \rightarrow \top, v_4 \rightarrow \top]$. This is equivalent to the fact that $\mu = (l_1!m, l_2!m)$ is consumed exactly by the execution of i from Fig.8 when $i_{\overline{v_1}}$ is chosen over i_{v_1}, i_{v_2} over $i_{\overline{v_2}}$, i_{v_3} over $i_{\overline{v_3}}$, and i_{v_4} over $i_{\overline{v_4}}$. For any such 3-CNF formula $\phi = C_1 \wedge C_2$ defined over $V = \{v_1, \cdots, v_4\}$, the 1-in-3-SAT problem can therefore be reduced to that of the membership of $(l_1!m, l_2!m)$ w.r.t. the interaction i constructed from ϕ as above.

As explained earlier, this sketch of proof can be extended to include any numbers p and q of resp. variables and clauses. It suffices to consider q lifelines $l_1, \cdots, l_q$, the multi-trace $\mu = (l_1!m, \cdots, l_q!m)$ and p parallelized sub-interactions $alt(i_{v_1}, i_{\overline{v_1}}), \cdots, alt(i_{v_p}, i_{\overline{v_p}})$.

Given that we have identified a case of multi-trace membership equivalent to an NP-complete problem, by reduction, multi-trace membership is NP-hard.

5 TOOL SUPPORT

Our prototype tool, called HIBOU is available online[4]. The inductive structure of definitions from Sec.3 facilitated their transcription into executable code (Rust language) for implementing HIBOU. The ω function is also implemented by building on-the-fly the sub-graph originating from the application of rule $R3$. As a result, HIBOU provides two main functionalities (1) multi-trace membership analysis and (2) interaction model exploration.

HIBOU can analyze multi-traces against interaction models and return a verdict. In addition to the *Pass* and *Fail* verdicts defined in Sec.4.2 (which state whether or not a multi-trace is accepted), HIBOU can also return a $WeakPass$ verdict for identifying prefixes of accepted multi-traces. Moreover, HIBOU expands the analysis to multi-traces defined over co-localizations that are not reduced to singletons. One can analyze $\mu = (\{a, b\} \rightarrow a!m.b!m, \{c\} \rightarrow c?m)$ defined over a co-localized sub-system composed of lifelines a and b, and another sub-system composed of lifeline c. In case where all lifelines form a single co-localization, our approach then corresponds to global trace analysis from [16].

Multi-trace analysis relies on the traversal of a tree constituted by nodes (i, μ). In practice, this traversal is interrupted when a Cov verdict (or $TooShort$, if the goal of the analysis is $WeakPass$) is reached. Various heuristics can be configured to quicken the process. HIBOU allows the use of Depth First Search (DFS) and Breadth First Search (BFS) as well as a way to set priority levels for the evaluation of certain types of actions.

Interaction model exploration uses χ to explore the semantics of interactions by unfolding paths $i \overset{*}{\rightsquigarrow} i'$. The computation of those paths can be stopped by setting filters on the size of the exploration tree (maximum depth, maximum number of nodes, etc.).

In HIBOU, traceability for end-users is facilitated given that we have access to the successions of nodes and can therefore draw analysis / exploration trees as illustrated in Fig.7.

[4]https://github.com/erwanM974/hibou_label

6 CONCLUSION

We have proposed an approach to decide on the membership of multi-traces w.r.t. a semantics defined on interaction models. The analysis consists in applying non-deterministic reading of the multi-trace using small-steps of the operational semantics. This approach has been validated with a formal proof of correctness using Coq, and a study on complexity. Moreover, a prototype tool that implements this analysis method has been developed in line with theoretical claims. Finally, as future work, we plan to exploit membership analysis to test distributed systems where logging of multi-traces is performed under observability limitations.

REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. 2001. Realizability and Verification of MSC Graphs. In *Automata, Languages and Programming, 28th International Colloquium, ICALP*, Vol. 2076. Springer, 797–808.

[2] C. Andrés, M. Cambronero, and M. Núñez. 2010. Formal Passive Testing of Service-Oriented Systems. In *IEEE International Conference on Services Computing, SCC*. IEEE, 610–613.

[3] B. Bannour, C. Gaston, and D. Servat. 2011. Eliciting Unitary Constraints from Timed Sequence Diagram with Symbolic Techniques: Application to Testing. In *18th Asia Pacific Software Engineering Conference, APSEC*. IEEE, 219–226.

[4] A. Bauer and Y. Falcone. 2016. Decentralised LTL monitoring. *Formal Methods Syst. Des.* (2016), 46–93.

[5] N. Benharrat, C. Gaston, R. M. Hierons, A. Lapitre, and P. Le Gall. 2017. Constraint-Based Oracles for Timed Distributed Systems. In *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS*, Vol. 10533. Springer, 276–292.

[6] J. A. Bergstra, I. Bethke, and A. Ponse. 1994. Process Algebra with Iteration and Nesting. *Comput. J.* 37, 4 (1994), 243–258.

[7] H. Dan and R. M. Hierons. 2011. Conformance Testing from Message Sequence Charts. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST*. IEEE, 279–288.

[8] H. Dan and R. M. Hierons. 2014. The Oracle Problem When Testing from MSCs. *Comput. J.* 57, 7 (2014), 987–1001.

[9] N. Dershowitz and J. P. Jouannaud. 1990. Handbook of Theoretical Computer Science (Vol. B). MIT Press, Chapter Rewrite Systems, 243–320.

[10] B. Genest and A. Muscholl. 2008. Pattern Matching and Membership for Hierarchical Message Sequence Charts. *Theory Comput. Syst.* 42, 4 (2008), 536–567.

[11] ITU. [n. d.]. Message Sequence Chart (MSC). http//www.itu.int/rec/T-REC-Z.120

[12] A. Knapp and T. Mossakowski. 2017. UML Interactions Meet State Machines - An Institutional Approach. In *7th Conference on Algebra and Coalgebra in Computer Science, CALCO (LIPIcs)*, Vol. 72. Schloss Dagstuhl, 15:1–15:15.

[13] L. Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, D. Malkhi (Ed.). ACM, 179–196.

[14] D. Longuet. 2012. Global and local testing from Message Sequence Charts. In *Proceedings of the ACM Symposium on Applied Computing, SAC*. ACM, 1332–1338.

[15] J. Mace, R. Roelke, and R. Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*. ACM, 378–393.

[16] E. Mahe, C. Gaston, and P. Le Gall. 2020. Revisiting Semantics of Interactions for Trace Validity Analysis. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE*, Vol. 12076. Springer, 482–501.

[17] F. Neves, N. Machado, and J. Pereira. 2018. Falcon: A Practical Log-Based Analysis Tool for Distributed Systems. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. IEEE, 534–541.

[18] H. N. Nguyen, P. Poizat, and F. Zaïdi. 2012. Passive conformance testing of service choreographies. In *Proceedings of the ACM Symposium on Applied Computing, SAC*. ACM, 1528–1535.

[19] OMG. [n. d.]. Business Process Model and Notation (BPMN). http://www.bpmn.org

[20] OMG. [n. d.]. Unified Modeling Language. http://www.uml.org

[21] Z. Qiu, X. Zhao, C. Cai, and H. Yang. 2007. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web, WWW*. ACM, 973–982.

[22] T. J. Schaefer. 1978. The Complexity of Satisfiability Problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, R. J. Lipton, W. A. Burkhard, W. J. Savitch, E. P. Friedman, and A. V. Aho (Eds.). ACM, 216–226.

[23] K. Sen, A. Vardhan, G. Agha, and G. Rosu. 2004. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *26th International Conference on Software Engineering, ICSE*. IEEE, 418–427.