INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Online Abstraction of Distributed Executions

Thomas Gazagnaire, Loïc Hélouët, Claude Jard

N°5736

Octobre 2005

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5736--FR+ENG

Online Abstraction of Distributed Executions

Thomas Gazagnaire*, Loïc Hélouët†, Claude Jard‡

Systèmes communicants
Projet DistribCom

Rapport de recherche n ° 5736 — Octobre 2005 — 28 pages

Abstract: This report proposes an on-line compression algorithm for distributed executions. An execution is decomposed into atomic communication patterns. Events are clustered together according to the following policy: the initial partial order structure is preserved, and valid global states are detected in order to find a coherent global state if a site fails. Two versions of the algorithm have been developed. The first version is online and centralized and does not depend on execution order of any concurrent events. Moreover, if clusters are bounded in size, the memory size needed for infinite executions is also bounded. The second version is distributed and is done by piggybacking subset of sites knowledge into messages: under certain conditions, size of piggybacked messages is bounded and the algorithm can be effectively implemented.

Key-words: abstraction, compression, online distributed algorithm

(Résumé : tsvp)

* IRISA / Rennes 1, thomas.gazagnaire@irisa.fr
† IRISA / INRIA, loic.helouet@irisa.fr
‡ IRISA / ENS Cachan, claude.jard@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Abstraction à la volée d'exécutions distribuées

Résumé : Ce document présente une méthode de compression pour des traces d'exécution distribuée. L'exécution est décomposée en motifs ne coupant pas de messages, qui conservent la structure d'ordre partiel initiale, et qui permettent de retrouver un état global cohérent en cas de panne d'un des sites. Deux versions de l'algorithme sont présentées. D'une part une version centralisée fonctionnant à la volée : l'abstraction construite est unique, c'est à dire qu'elle ne dépend pas de l'ordre d'exécution des evénements concurrents. De plus, pour des exécutions infinies, la taille nécessaire à la réalisation de l'algorithme est bornée grâce à l'identification à la volée de préfixes qui ne seront jamais modifiés dans le futur. D'autre part une version distribuée qui fonctionne par estampillage des messages de l'exécution observée : sous certaine contraintes structurelles, les compexités temporelle et spatiale de cet algorithme sont bornées et l'algorithme est donc réellement implémentable.

Mots-clé : abstraction, compression, algorithmes à la volée et distribués

Contents

1 Introduction

Abstraction is often a key-point for the analysis and understanding of distributed systems executions. It may serve two objectives: the first one is to keep in a restricted vision of a system execution the only information needed to validate a property or exhibit faulty behaviors such as races on shared variables. The second objective is to exploit redundancy to compress executions, hence reducing the memory needed to store or study executions of a program.

So far, much attention has been paid to compression and abstraction of sequential programs executions, but very little work addresses the problem of abstraction for distributed systems. [3] proposes an adaptation of SEQUITUR [10] and of the WPP [7] for parallel programs communicating via shared variables. Roughly speaking, the executions are partitioned into strings of events executed by the same process. These strings are then compressed. [1] proposes several approaches to compress partially ordered strings. The main idea is to consider a labeled partial order as a string σ with a commutation relation I. σ and I define an equivalence class $[\sigma]$ of strings that are equivalent to σ up to commutation. The idea behind the algorithms of [1] is to provide a compression mechanism γ such that the decompressed string $\gamma^{-1}(\gamma(\sigma)) \in [\sigma]$. Abstraction of parallel program executions raises a new problem that does not exist for sequential programs: the compression rate for an online algorithm like SEQUITUR depends on the order of observation of events: two equivalent strings may have different compression rates, and a different compressed representation.

We propose a distributed online algorithm that computes an unique abstraction of executions for distributed systems communicating via asynchronous messages. The main idea is to decompose the execution on the fly into small subset of events, called atoms, without cuting messages. These atoms define a new alphabet, and the distributed execution is abstracted to a labeled partial order over this alphabet. Compression comes from the redundancy within segments of execution. An interesting property of the abstraction is that the decompression of any prefix of the partial order is a distributed control point [9] of the execution (i.e a global state without messages in transit). Note however that this abstraction only applies when the system's conception ensures that such control points appear from time to time: for such cases, the atom alphabet of any execution is finite and a system may not exhibit infinite atoms.

To be efficient, the abstraction algorithm has to satisfy 3 main criteria: First, it has to be **incremental**. For each computation step, we want to reduce time complexity and reuse what was already calculated in precedent steps. Then, it has to be **distributed**, i.e. each node will compute internal results and send it to other nodes. Furthermore, we want to reduce the use of the network channels, i.e. the size of information exchanged among nodes. The final requirement is to **match the communication architecture**, i.e. we do not want to add new messages or communication channels to the observed system, but only add additional data to already existing messages.

The report is organized as follows: section 2 introduces basic definitions on partial orders and distributed executions. Section 3 proposes an incremental but centralized algorithm

for executions decomposition. Section 4 proposes a distributed implementation of this algorithm. Section 5 concludes this work.

2 Executions, partial orders and abstraction

The architecture assumed in this work is a network of n nodes that communicate via asynchronous messages. This set of nodes can be mapped over a set of integers $[1..n] \subset \mathbb{N}$. A communication link between each pair of nodes does not need to exist. A supervisor collects information from the nodes of the network, and builds online an abstraction of the execution. Figure 1 shows the architecture of a supervised network of 6 nodes. Communication links between nodes are represented by a plain line. Nodes can also sent results of local observations to a supervisor. This supervisor is in charge of computing an abstract representation $\mathcal{A}(\mathcal{E})$ of the running execution $\mathcal{E}$. The runs of a system distributed over this kind of network can be represented as partial orders. The abstraction mechanism defined hereafter provides a coarse grain description of an execution as an ordering of phases of the underlying protocol. For partial orders, abstraction can be defined as a partition, providing a quotient partial order on important phases of a run rather than an ordering on a set of events. A possible abstraction is to consider atoms [5, 4], i.e. the smallest communication closed subsets of events. An offline abstraction algorithm called atomic decomposition was proposed in [5]. In this report, we propose an online, distributed and incremental approach of this abstraction.

2.1 Event Graph

We will consider an infinite set of events denoted by $\mathbb{E}$ that represents all computation steps that can occur on nodes of a network. An event is either an internal action, a message emission or reception, executed by a single node. This is modeled with a typing function $t : \mathbb{E} \to \{i, s, r\}$ that associates a type (internal, emission, reception) to each event. We denote by $\mathbb{E}_i$, $\mathbb{E}_s$ and $\mathbb{E}_r$ the corresponding subsets of $\mathbb{E}$. Similarly, we define a locality function $l : \mathbb{E} \to [1..n]$ that associates a specific site number to each event. Communications are defined as asynchronous messages, that is they are defined as pairs of emission and reception events. Furthermore, we suppose that there is an unique signature identifying each message sent on the network. So, we define a function $m : \mathbb{E}_s \to \mathbb{E}_r$ that pairs the emission and the reception of a message in $\mathbb{E}$. m is total and bijective, and for an arbitrary set E of events, the function m pairing emissions and receptions of E is unique. From now on, we will consider that the 4-tuple $(\mathbb{E}, l, t, m)$ is always defined. Moreover, we will say that an event set $E \subset \mathbb{E}$ is **communication closed** if $m(E \cap \mathbb{E}_s) \subseteq E \cap \mathbb{E}_r$ and $m^{-1}(E \cap \mathbb{E}_r) \subseteq E \cap \mathbb{E}_s$.

A description of a system execution is usually given not only in terms of subset of $\mathbb{E}$ but also in terms of causal precedence among the events. An **event graph** is a partial order $\mathcal{E} = (E, \leq)$, with $E \subset \mathbb{E}$ and $\leq \subset E \times E$, such that $\leq$ is reflexive, transitive, antisymmetric, and satisfies the two following axioms:

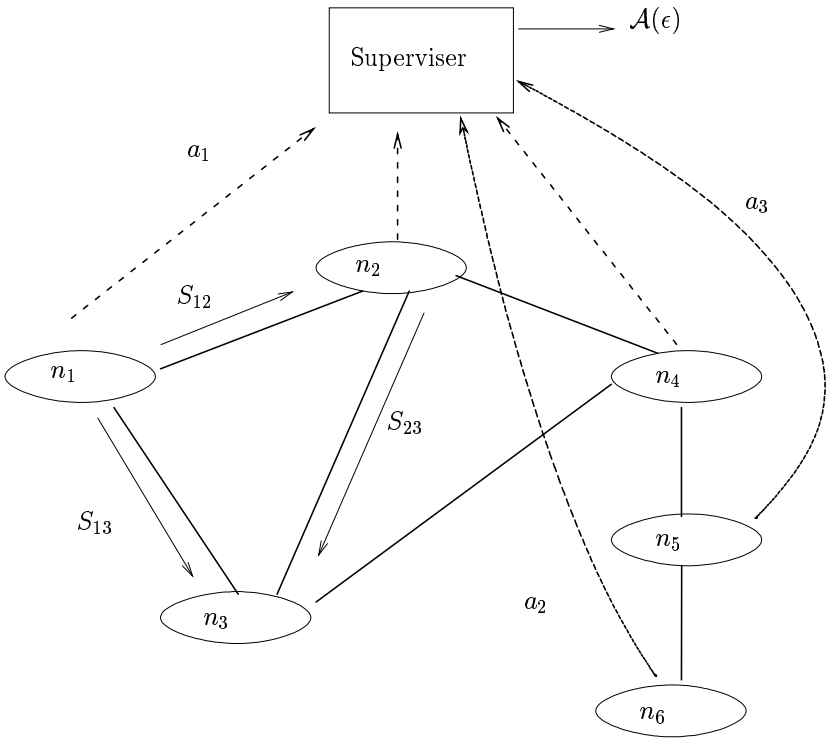


Figure 1: Architecture of the abstraction mechanism

(no self-concurrency) for all e, $e' \in E$, $l(e) = l(e') \Longrightarrow e \leq e' \vee e' \leq e$. i.e. all events executed on a node are comparable.

(message causality) for all e, $e' \in E$, if $e' = m(e)$ then $e \leq e'$. If $l(e) \neq l(e')$ and $e \leq e'$, then there exists a sequence $\left((e_s^i, e_r^i)\right)_{i\in[1..p]}$ belonging to $(\mathbb{E}_s \times \mathbb{E}_r)^p$ such that $m(e_s^i) = e_r^i$, $l(e_r^i) = l(e_s^{i+1})$, and $e \leq e_s^1 \leq e_r^1 \leq e_s^2 ... \leq e_r^p \leq e'$; In other words, if two events located on different nodes n, n' are comparable, then there is a sequence of messages from n to n' that imposes this causal precedence.

Very often in distributed systems, we use the additional assumption that messages on the same channel do not overtake: if $m(e_1) = e_1'$, $m(e_2) = e_2'$, $l(e_1) = l(e_2)$, and $l(e_1') = l(e_2')$ then $e_1 \leq e_2 \Leftrightarrow e_1' \leq e_2'$. This assumption is not essential in our framework. Note however that when this property holds, message emissions and receptions can be paired without need for a unique message signature.

Let $\mathcal{E} = (E, \leq)$ be an event graph. The **past** of an event e is denoted by $\downarrow_{\mathcal{E}}(e)$ and is the descending cone from e, i.e. $\downarrow_{\mathcal{E}}(e) = \{e' \in E \mid e' \leq e\}$. Similarly, we can define the **future** of an event e by $\uparrow_{\mathcal{E}}(e) = \{e' \in E \mid e \leq e'\}$. Let us denote by $E_i = \{e \in E \mid l(e) = i\}$. Then, $\mathcal{E}' = (E', \leq')$ is a **prefix** (resp. a **suffix**) of $\mathcal{E}$ if and only if $E' \subseteq E, \leq' = \leq \cap E' \times E'$ and $\downarrow_{\mathcal{E}}(E') = E'$ (resp. $\uparrow_{\mathcal{E}}(E') = E'$). Finally we will denote by $max_i(\mathcal{E})$ (resp. $min_i(\mathcal{E})$) the **maximum** (resp. **minimum) event on node** i, i.e. $max_i(\mathcal{E}) = max\{e \in E \mid l(e) = i\}$ (resp. $min_i(\mathcal{E}) = min\{e \in E \mid l(e) = i\}$).

Let $\mathcal{E} = (E, \leq)$ and $\mathcal{E}' = (E', \leq')$ be two event graphs. The **sequential composition** of $\mathcal{E}$ and $\mathcal{E}'$ is denoted by $\mathcal{E} \circ \mathcal{E}'$ and is the event graph: $\mathcal{E}_c = (E \cup E', (\leq \cup \leq' \cup \leq_l \cup \leq_m)^*)$, where:

- $\leq_l = \{(max_i(\mathcal{E}), min_i(\mathcal{E}'))\}_{i\in[1..n]}$. This relation is sufficient to avoid self-concurrency.
- $\leq_m = \{(e, m(e)) \mid (e \in E \cap \mathbb{E}_s \wedge m(e) \in E') \vee (e \in E' \cap \mathbb{E}_s \wedge m(e) \in E)\}$. This relation builds the unique message relation between events of E and E'.

We will say that a sequential composition between $\mathcal{E}_1$ and $\mathcal{E}_2$ is **well-defined** if $\mathcal{E}_1$ and $\mathcal{E}_2$ have disjoint event sets and $\mathcal{E}_1 \circ \mathcal{E}_2$ has no cycle (this is the usual notion of well-defined sequential composition for partial orders that is used for example, to merge Message Sequence Charts [6]). Let us denote by $\mathbb{B} = \{(E, \leq^*) \mid E \in \mathcal{P}(\mathbb{E}) \wedge \leq \subseteq E \times E\}$ the set of all event graphs ($\mathcal{P}(\mathbb{E})$ is the set of all subsets of $\mathbb{E}$).

A **cut** is a function from $\mathbb{B}$ to $\mathbb{B} \times \mathbb{B}$: $c(\mathcal{E}) = (\mathcal{E}_1, \mathcal{E}_2)$ such that $\mathcal{E}_1$ is a prefix of $\mathcal{E}$, $\mathcal{E}_2$ is a suffix of $\mathcal{E}$, $E_1 \cap E_2 = \emptyset$,$E_1 \neq \emptyset$,$E_2 \neq \emptyset$ and $E_1 \cup E_2 = E$. We will denote by $\mathcal{E}_1 = c_1(\mathcal{E})$ and $\mathcal{E}_2 = c_2(\mathcal{E})$. We can also decompose an event graph $\mathcal{E}$ into a set of singleton graphs $\{(\{e\}, \emptyset) \mid e \in E\}$ with a finite number of successive cuts of the form $c_1, \ldots c_{|E|-1}$, where each c_i is applied to a non trivial component obtained after preceding cuts. To simplify notations, we will denote by $c^k(\mathcal{E})$ an application of $k-1$ successive cuts that partitions $\mathcal{E}$ into k components $c_1^k(\mathcal{E}), c_2^k(\mathcal{E}), \ldots c_k^k(\mathcal{E})$. Obviously, if we denote by e_i the event appearing in singleton $c_i^{|E|}(\mathcal{E})$, $e_1 \ldots e_{|E|}$ is a linearization of $\mathcal{E}$.

Proposition 2.1 *For any cut c, we have $\circ.c = Id_{\mathbb{B}}$, i.e. for all event graph $\mathcal{E} \in \mathbb{B}$, $\mathcal{E} = c_1(\mathcal{E}) \circ c_2(\mathcal{E})$*

Proof:

We want to show that $\circ.c = Id_{\mathbb{B}}$, ie. $\circ.c : \mathbb{B} \xrightarrow{c} \mathbb{B} \times \mathbb{B} \xrightarrow{\circ} \mathbb{B}$ is identity. Let us note $c_1(\mathcal{E}) = \mathcal{E}_1$ and $c_2(\mathcal{E}) = \mathcal{E}_2$. We will work with the covering relation $\prec_{\mathcal{E}}$ of the partial order $\leq (\prec_{\mathcal{E}} = \{(e, e') \in \leq | \nexists e'', e \neq e'' \wedge e' \neq e'' \wedge e \leq e'' \leq e'\})$ and show that this relation is the same in $\mathcal{E}_1 \circ \mathcal{E}_2$ and in $\mathcal{E}$. Let us consider two events e and e' in E. Two cases may appear:

- First, if $e \prec_{\mathcal{E}} e'$: if e and e' are in the same component i after the cut, then $e \prec_{\mathcal{E}_i} e'$, and as $\circ$ do not remove causality $e \prec_{\mathcal{E}_1 \circ \mathcal{E}_2} e'$. If they are not in the same component, they were either on the same instance or they were separated by a message. And $\circ$ adds the good causality we want (ie. $\leq_m$ and $\leq_l$).
- Second, if $e \not\prec_{\mathcal{E}} e'$, $e \not\prec_{\mathcal{E}_i} e'$ because the cut does not add causalities. Then, as $e \not\prec_{\mathcal{E}} e'$, e and e' are not respectively the emission and the reception of a message, so $\leq_m$ do not contains (e, e') or (e', e). Similarly, if $l(e) = l(e')$, as $e \not\prec_{\mathcal{E}} e'$ then there is a e'' such that $l(e'') = l(e)$, $e \prec_{\mathcal{E}} e''$ and $e'' \leq e'$. Furthermore, $e'' \in \mathcal{E}_1$ or $e'' \in \mathcal{E}_2$. So, $(e, e') \notin \leq_l$. Thus, $e \not\prec_{\mathcal{E}_1 \circ \mathcal{E}_2} e'$.

□

This property means that sequential composition can be used to rebuild the cut event graphs. So, if we use cuts to decompose a global system execution into smaller parts to study them separately, we can always rebuild the initial execution using sequential composition. This allows us to focus first on subsets of global traces before assembling them.

2.2 Abstract graph

Let $\mathcal{E} = (E, \leq)$ be an event graph. We want to build an abstraction of $\mathcal{E}$ called an **abstract graph**, i.e. partition E into subsets of events grouped according to a given property, and find an ordering among these subsets. More formally, for an event graph $\mathcal{E}$ and a equivalence relation $\mathcal{R}_{\mathcal{E}}$ among events of $\mathcal{E}$, we are looking for the quotient graph $\mathcal{E}/\mathcal{R}_{\mathcal{E}}$. The abstraction proposed in this paper is atomic decomposition: we are looking for the finest abstraction which does not separate emissions and receptions of a message and which defines an acyclic quotient graph. The equivalence relation that characterizes the properties of atomic decomposition is $e\mathcal{R}_{\mathcal{E}}e' \Leftrightarrow e \leq_{\mathcal{R}_{\mathcal{E}}} e' \wedge e' \leq_{\mathcal{R}_{\mathcal{E}}} e$, where: $x \leq_{\mathcal{R}_{\mathcal{E}}} y$ if and only if: $x \leq y$ or $x = m(y)$ or $\exists z, x \leq_{\mathcal{R}_{\mathcal{E}}} z \leq_{\mathcal{R}_{\mathcal{E}}} y$.

Each class of the quotient is the union all equivalent events. The quotient order $\mathcal{E}/\mathcal{R}_{\mathcal{E}}$ computed is thus of the form $\mathcal{A} = (A, \alpha)$, with A a partition of E, ie. $\forall a \in A, (\ e \in a \wedge\ e' \in a) \Leftrightarrow (e\mathcal{R}_{\mathcal{E}}e')$ and α the quotient relation of $\leq$ over A. The elements of A are often called the **atoms** of $\mathcal{E}$. A polynomial algorithm to compute atoms in Message Sequence Charts was proposed in [5].

In the rest of the paper, we will denote by $\mathcal{A}(\mathcal{E})$ the unique abstract graph associated with the event graph $\mathcal{E}$, and by $\mathbb{A} = \{\mathcal{E}/\mathcal{R}_\mathcal{E} \mid \mathcal{E} \in \mathbb{B}\}$, the set of all abstract graphs. Let us now define sequential composition of abstract graphs. It is similar to the composition of event graphs, except that we may add new dependencies among components and that we have to ensure that the result obtained remains acyclic.

Let $\mathcal{A} = (A, \alpha)$ and $\mathcal{A}' = (A', \alpha')$ be two abstract graphs with disjoint event sets, i.e. $\bigcup_{a\in A} a \cap \bigcup_{a'\in A'} a' = \emptyset$. The sequential **pre-composition** of $\mathcal{A}$ and $\mathcal{A}'$, denoted by $\mathcal{A} \circ_p \mathcal{A}'$ is the graph $(A \cup A', (\alpha \cup \alpha' \cup \alpha_l \cup \alpha_m)^*)$, where:

- $\alpha_l = \{(max_i(\mathcal{A}), min_i(\mathcal{A}')) \mid i \in [1..n]\}$, where $max_i(\mathcal{A}) = max(\{a \mid i \in l(a)\}$. α_l is the minimal causal relation that avoids self-interference in abstract event graphs.
- $\alpha_m = \{(a, a') \in A \times A' \cup A' \times A \mid (m(a) \cap a' \neq \emptyset \vee m(a') \cap a \neq \emptyset)\}$ is the causal relation needed to keep emission and reception of a message in the same strongly connected component.

The pre-composition of two abstract graphs is not a partial order (it may still contain cycles). The **sequential composition** merges vertices of connected components of a pre-composition. Let $\mathcal{A}_p = (A_p, \alpha_p) = \mathcal{A} \circ_p \mathcal{A}'$ be the graph obtained after a pre-composition of $\mathcal{A}$ and $\mathcal{A}'$. The sequential composition of $\mathcal{A}$ and $\mathcal{A}'$ is denoted by $\mathcal{A} \circ \mathcal{A}'$ and is the quotient graph of $\mathcal{A}_p$ w.r.t. strongly connected components in $\mathcal{A}_p$. In other words, if we denote by $\mathcal{CC}$ the equivalence relation among elements of strongly connected components of a graph, $\mathcal{A} \circ \mathcal{A}' = \mathcal{A}_p/\mathcal{CC}$. Again, each class of the quotient by $\mathcal{CC}$ is the union of all equivalent events.

An algorithm to compute the composition of two abstract graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ is then straightforward. The first step is to compute a pre-composition of $\mathcal{A}_1$ and $\mathcal{A}_2$. Using their cover relations and an appropriate data structure, this can be done in $O(|A_1| + |A_2|)$. The second step of the composition consists in finding connected components in $\mathcal{A}_1 \circ_p \mathcal{A}_2$. This can be performed in linear time (in the size of $\mathcal{A}_1 \cup \mathcal{A}_2$) using Tarjan's algorithm [12].

Proposition 2.2 *Let $\mathcal{E}$ be an event graph, and c be a cut. Then, we have $\mathcal{A}(\mathcal{E}) = \mathcal{A}(c_1(\mathcal{E})) \circ \mathcal{A}(c_2(\mathcal{E}'))$.*

Proof:

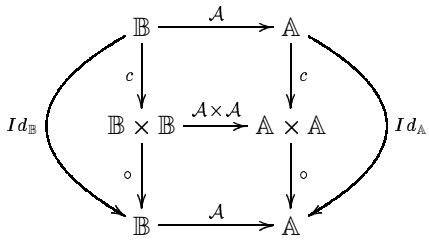
We want to show that $\mathcal{A}(\mathcal{E}) = \mathcal{A}(c_1(\mathcal{E}) \circ c_2(\mathcal{E})) = \mathcal{A}(c_1(\mathcal{E})) \circ \mathcal{A}(c_2(\mathcal{E}))$. Let us note $\mathcal{R}$ the equivalence relation used to build the an abstract graph from $\mathcal{E}$, i.e. $\mathcal{A}(\mathcal{E}) = \mathcal{E}/\mathcal{R}$. Let us note $\mathcal{R}_1$ and $\mathcal{R}_2$ the similar equivalence relations on $\mathcal{E}_1 = c_1(\mathcal{E})$ and $\mathcal{E}_1 = c_1(\mathcal{E})$, and $\mathcal{R}'$ the equivalence relation such that $\mathcal{A}(c_1(\mathcal{E}_1)) \circ \mathcal{A}(c_2(\mathcal{E}_2)) = (\mathcal{E}_1/\mathcal{R}_1) \circ_p (\mathcal{E}_2/\mathcal{R}_2)/\mathcal{CC} = (\mathcal{E}_1 \circ_p \mathcal{E}_2)/\mathcal{R}'$. We will also note $e \leq_p e'$ if $e \in a$, $e' \in a'$ and $a\alpha_p a'$. Let us show that $\mathcal{R} = \mathcal{R}'$.

- First, if we have $x \leq_\mathcal{R} y \leq_\mathcal{R} x$ and $y \leq_m z$ or $y \leq_l z \leq_\mathcal{R} x$ then $x \leq_\mathcal{R} z \leq_\mathcal{R} x$ as $\leq_l$ is a subset of $\leq$ and $x \leq_m z$ means $x = m(z)$ or $z = m(x)$. Thus, $x \leq_\mathcal{R} y \leq_{m,l} z \leq_\mathcal{R} x$ implies $x\mathcal{R}y\mathcal{R}z$.

- Second, let a_1 and a_2 be two event classes merged by $\mathcal{CC}$. It means that x_1, x_2, y_1, y_2 exist such that x_1, y_1 are in a_1, x_2, y_2 are in a_2, $x_1 \leq_p x_2$ and $y_2 \leq_p x_2$. So, for each x in a_1 and y in a_2 we will have: $x \leq_{\mathcal{R}_1} x_1 \leq_p x_2 \leq_{\mathcal{R}_2} y \leq_{\mathcal{R}_2} y_2 \leq_p y_1 \leq_{\mathcal{R}_1} x$. As $\mathcal{R}_1 \subset \mathcal{R}$ and $\mathcal{R}_2 \subset \mathcal{R}$, we have $x \leq_{\mathcal{R}} x_1 \leq_{\mathcal{R}} x_2 \leq_{\mathcal{R}} y \leq_{\mathcal{R}} y_2 \leq_{\mathcal{R}} y_1 \leq_{\mathcal{R}} x$, i.e. $x \leq_{\mathcal{R}} y \leq_{\mathcal{R}} x$ and $x\mathcal{R}y$. Thus, $\mathcal{R}' \subset \mathcal{R}$.
- Third, let a be in $\mathcal{A}(\mathcal{E})$. If $a \cap A_1 = \emptyset$ or $a \cap A_2 = \emptyset$ then a is in $\mathcal{A}(c_1(\mathcal{E})) \circ \mathcal{A}(c_2(\mathcal{E}))$ because $\mathcal{R}_{|A_i} = \mathcal{R}_i$. Else, we can choose x in $a \cap A_1$ and y in $a \cap A_2$ such that $x \leq_{\mathcal{R}} y \leq_{\mathcal{R}} x$. Then, we can find x_1, y_1 in $a \cap A_1$ and x_2, y_2 in $a \cap A_2$ such that $x \leq_{\mathcal{R}} x_1 \prec_{\mathcal{R}} x_2 \leq_{\mathcal{R}} y \leq_{\mathcal{R}} y_2 \prec_{\mathcal{R}} y_1 \leq_{\mathcal{R}} x$. $x_1 \prec_{\mathcal{R}} x_2$ means either $x_1 = m(x_1)$ or $x_1 \prec x_2$ and there exists i such that $x_1 = max_i(a \cap A_1)$ and $x_2 = min_i(a \cap A_2)$. Thus $x_1 \leq_l x_2$. And finally, we have: $x \leq_{\mathcal{R}_1} x_1 \leq_p x_2 \leq_{\mathcal{R}_2} y \leq_{\mathcal{R}_2} y_2 \prec_p x_2 \leq_{\mathcal{R}_1} x$. So, x and y are in a cycle in $\mathcal{A}(c_1(\mathcal{E})) \circ_p \mathcal{A}(c_2(\mathcal{E}))$. Thus $\mathcal{R} \subset \mathcal{R}'$.

□

To summarize, we obtain the following commutative diagram:



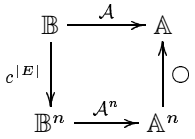
Using proposition 2.2, we can easily obtain the following decomposition theorem.

Theorem 2.1 *Let $\mathcal{E} = (E, \leq)$ be an event graph, and let $c^{|E|}$ be a decomposition of $\mathcal{E}$ into singletons. Then we have $\mathcal{A}(\mathcal{E}) = \bigcirc_{i \in 1..|E|} \mathcal{A}(c_i^{|E|}(\mathcal{E}))$, where $\bigcirc_{i \in 1..k} \mathcal{A}_i = \mathcal{A}_1 \circ \mathcal{A}_2 \ldots \mathcal{A}_k$*

Proof:
This theorem easily follows from the recursive application of proposition 2.2.
□

Theorem 2.1 shows that abstraction of an event graph is unique (up to isomorphism). As the abstraction built does not depend on the linearization chosen to compose singletons, this uniqueness is preserved even when events of an execution are observed online. This is an interesting property for an online compression algorithm, as it guarantees a fixed compression rate for a given execution. Note that usually, compression rates for algorithms like SEQUITUR may depend on the order of occurrence between concurrent events, as shown

in [1]. To summarize, we obtain the commutative diagram:



Note that the size of an execution can be very large, and that it is not always possible to keep a complete event graph in memory. Abstraction can help reducing the size of computations, but is of course subject to the same limits. Hence, working offline on a complete execution or even on its abstraction rapidly imposes limits to the size of executions that can be treated. However, when abstraction in computed online, some parts of an abstraction are not modified in the future. Hence, once they have been detected and stored, they become useless, and can be forgotten. These parts of abstract graphs will be called stable.

Let $\mathcal{A} = (A, \alpha)$ be an abstract graph. $\mathcal{A}$ is **stable** if and only if for all $\mathcal{A}'$, for all $a \in \mathcal{A}$, a is also an abstract class of $\mathcal{A} \circ \mathcal{A}'$. We will denote by $\mathbb{A}^s$ the set of all stable abstract graphs. A **prefix** of an abstract graph $\mathcal{A}$ is an abstract graph $\mathcal{A}' = (A', \alpha')$ such that $A' \subseteq A$, $\alpha' = \alpha \cap A' \times A'$ is the restriction of α to A', and $\downarrow_{\mathcal{A}} (A') = A'$. Event classes of stable prefix are maximal, i.e. we will not add new events to these classes. The **maximal stable prefix** of $\mathcal{A}$ is a stable prefix $P_{\mathcal{A}}$ maximal by inclusion, i.e. for all prefix P of $(\mathcal{A} - P_{\mathcal{A}})$, $P_{\mathcal{A}} \circ P$ is not a stable prefix anymore.

Proposition 2.3 *The maximal stable prefix of an abstract graph $\mathcal{A}$ is exactly the largest communication closed prefix of $\mathcal{A}$.*

Proof:

We want to show that $P_{\mathcal{A}}$, the biggest communication closed prefix of $\mathcal{A}$ is the maximal stable prefix of $\mathcal{A}$.

First, let us show that a non communication closed prefix is not stable. This means that there exists an e in $a \in P_{\mathcal{A}}$ such that $m(e)$ (or $m^{-1}(e)$) $\notin \bigcup_{a \in A}$. Then let us compose $\mathcal{A}$ with $(\{m(e)\}, \emptyset)$. $P_{\mathcal{A}}$ is modified, because it now contains $a \cup \{m(e)\}$. Thus, stable prefixes are communication closed.

Second, let us show that communication closed prefix are stable. For any $\mathcal{A}''$ composed with $P_{\mathcal{A}}$ we will add only one kind of causal dependency: local causalities, i.e. causal dependencies $e \leq e'$ such that $l(e) = l(e')$. Indeed, as $P_{\mathcal{A}}$ is communication closed, $\alpha_{m|P_{\mathcal{A}}}$ will always be empty. But by adding only local causalities, we cannot create cycles. Moreover, as $P_{\mathcal{A}}$ is a prefix it is closed by causal precedence, and no message causality can be added before $P_{\mathcal{A}}$. So $P_{\mathcal{A}}$ still remains the same.

Finally, by cardinality we can conclude that the maximal stable prefix is exactly the largest communication closed prefix.

□

Proposition 2.3 illustrates another nice property of abstract graphs: all stable prefixes are closed by communication. Hence, stable prefixes define consistent distributed control points. A distributed control point (or snapshot) [9] is a collection of local states (one per node in the system) also called checkpoints. Netzer et Al. showed that a snapshot is consistent if and only if all its checkpoints are concurrent. As our abstract graphs are built on atoms, this ensures that the event graph obtained from any prefix of a stable abstract graph does not contain messages that are received but not sent which is an essential property for snapshots consistency. Note that stable abstract graphs do not define all consistent snapshots of an execution as they do not contain messages that are sent but not received which is allowed in snapshots.

As for event graphs, a cut of an abstract graph $\mathcal{A} = (A, \alpha)$ is defined as a partition $c : \mathbb{A} \longrightarrow \mathbb{A} \times \mathbb{A}$ of $\mathcal{A}$ into a prefix $c_1(\mathcal{A})$ and suffix $c_2(\mathcal{A})$. A **stable cut** is a cut c^s from $\mathbb{A}$ to $\mathbb{A}^s \times \mathbb{A}$ such that $c_1^s(\mathcal{A})$ is a stable prefix of $\mathcal{A}$.

Proposition 2.4 *For any stable cut c^s, we have $\circ_p.c^s = \circ.c^s = Id_{\mathbb{A}}$, i.e. for all abstract graph $\mathcal{A} \in \mathbb{A}$, $\mathcal{A} = c_1^s(\mathcal{A}) \circ_p c_2^s(\mathcal{A}) = c_1^s(\mathcal{A}) \circ c_2^s(\mathcal{A})$.*

Proof:

We want to show that $c_1^s(\mathcal{A}) \circ_p c_2^s(\mathcal{A}) = c_1^s(\mathcal{A}) \circ c_2^s(\mathcal{A})$ and that $c_1^s(\mathcal{A}) \circ_p c_2^s(\mathcal{A}) = \mathcal{A}$.

First, as $c_1^s(\mathcal{A})$ is stable, then for all $a \in c_1^s(\mathcal{A})$, a is also a class of $\mathcal{A}$. So, equivalence classes in $c_2^s(\mathcal{A})$ are also preserved by $\circ_p$ and no cycle is created during composition. Thus $c_1^s(\mathcal{A}) \circ_p c_2^s(\mathcal{A}) = c_1^s(\mathcal{A}) \circ c_2^s(\mathcal{A})$.

Second, for each stable cut of an abstract graphs, there is an unique cut of event graphs, which partitions events in the same two components. So $c_1^s(\mathcal{A}(\mathcal{E})) \circ_p c_2^s(\mathcal{A}(\mathcal{E})) = \mathcal{A}(c_1^s(\mathcal{E})) \circ_p \mathcal{A}(c_2^s(\mathcal{E}))$ and then with proposition 2.2 we finally have $c_1^s(\mathcal{A}(\mathcal{E})) \circ_p c_2^s(\mathcal{A}(\mathcal{E})) = \mathcal{A}(\mathcal{E})$ which is what we want.

□

This property is useful to reduce the complexity of sequential composition. To compose a stable abstract graph $\mathcal{A}$ with an abstract graph $\mathcal{A}'$, we do not need to compute α_m in the precomposition, nor to quotient the result of the pre-composition of $\mathcal{A}$ and $\mathcal{A}'$.

3 Centralized abstraction algorithm

A direct consequence of proposition 2.3 is that to compute the stable part of an abstract graph we just have to search the largest communication closed prefix (which can be done in linear time). Moreover, a stable prefix can be computed in an incremental way (a part of an event graph that has been declared stable can not become unstable after concatenating it with any event graph). This property will be used to reduce the complexity of computations (a stable prefix will not be considered anymore in the abstract graph construction) and the state space used. The following algorithm computes a decomposition of $\mathcal{A}_1 \circ \mathcal{A}_2$ into a

Algorithm 1 Prefix incremental computation

incr_compose: $\mathbb{A}^s \times \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A}^s \times \mathbb{A}$

1: Input: $P_{\mathcal{A}_1}, \mathcal{A}'_1, \mathcal{A}_2$ with $P_{\mathcal{A}_1}$ stable, $\mathcal{A}'_1, \mathcal{A}_2$ potentially unstable.
2: $\mathcal{A} = \mathcal{A}'_1 \circ \mathcal{A}_2$
3: $P = \max\_stable\_prefix(\mathcal{A})$
4: $P_{\mathcal{A}_3} = P_{\mathcal{A}_1} \circ_p P$
5: $\mathcal{A}'_3 = \mathcal{A} - P$
6: Output: $P_{\mathcal{A}_3}, \mathcal{A}'_3$

maximal stable prefix $P_{\mathcal{A}_3}$ and a potentially unstable part $\mathcal{A}'_3$, for a given decomposition of $\mathcal{A}_1$ into its maximal stable prefix $P_{\mathcal{A}_1}$ and a potentially unstable part $\mathcal{A}'_1$.

As $P_{\mathcal{A}_3} \S \mathcal{A}' P_{\mathcal{A}_1} \circ \mathcal{A}'_1 \circ \mathcal{A}_2$, this algorithm provides a correct decomposition of $\mathcal{A}_1 \circ \mathcal{A}_2$. Moreover, $P_{\mathcal{A}_3}$ belongs to $\mathbb{A}^s$ because $P_{\mathcal{A}_1} \circ_p P$ is a composition of 2 stable abstract graphs. As composition of stable graphs is a simplified precomposition, the complexity of the algorithm depends more on the size of $\mathcal{A}'_1$ and $\mathcal{A}_2$ than on that of $P_{\mathcal{A}_1}$. Hence, to ensure efficient calculus, the point is to increase the size of $P_{\mathcal{A}_1}$ to reduce the time needed at different stages of abstraction. Note also that we do not need to remember the event graph from which the stable abstract graphs are generated to compose them. Hence stable prefixes are also a way to use less memory during abstraction.

Before distributing an abstraction mechanism over a given architecture, let us introduce a centralized incremental version of our algorithm. This algorithm is executed on a supervisor node that monitors an architecture where nodes do not communicate intermediate results to each other, and signal the occurrence of every event to the supervisor. The abstract graph of an event graph $\mathcal{E}$ is built step by step by successive addition of a singleton abstract graph $(\{\{e\}\}, \emptyset)$ to the existing graph $\mathcal{A}$, where e is a minimal element in $\mathcal{E}/(\bigcup_{a \in \mathcal{A}} a)$. The choice of a singleton simulates the occurrence of an event in a distributed execution.

Algorithm 2 Abstract Graph Computation

abstract: $\mathbb{B} \rightarrow \mathcal{A}^s \times \mathcal{A}$

1: Input: $\mathcal{E}$
2: $\mathcal{A} = (\emptyset, \emptyset)$; $P = (\emptyset, \emptyset)$
3: **while** $E \neq \emptyset$ **do**
4: Choose $e \in min(\mathcal{E})$
5: $E = E - \{e\}$
6: $(P, \mathcal{A}) = \text{incr-compose}(P, \mathcal{A}, (\{\{e\}\}, \emptyset))$
7: **end while**
8: Output: $P, \mathcal{A}$

From theorem 2.1, we know that if $|\mathcal{E}| = m$ we can decompose $\mathcal{E}$ into m singleton event graphs with successive cuts, and recompose them with $\circ$. The choice of a minimal event

in this algorithm is a cut, which guarantees that the incremental version of the algorithm produces the correct abstraction $\mathcal{A}(\mathcal{E})$.

4 Distributed algorithm

Let us now define a distributed version of the abstraction algorithm. Our network model consists of n nodes that communicate via asynchronous messages. We suppose that nodes cannot fail (they continually produce observable events). Furthermore, we assume that communications can not be delayed for an unbounded time.

All nodes will execute the same algorithms to build a coherent abstraction of an execution in a distributed way. Each node performs a local computation of the abstraction according to its knowledge of the running execution, and communicates its results to other sites. The abstraction algorithm uses the existing communication means: it does not add new communication links or new messages to the existing architecture. Transfer of local computations is done by piggy-backing information in existing messages. Upon reception of a messages from a distant site, a node will update its knowledge on currently built atoms. The goal of each node is to reduce the state space used during abstraction and to bound piggy-backed message size. Whenever an atom is stable (i.e. it is an atom contained in a stable prefix) and is useless for abstraction on all nodes, it is sent to a supervisor, and removed from memory.

4.1 Parallel composition

Two different nodes may compute their own vision of the same part of an execution. Upon reception of a message, a node has to update its own information, but should take into account redundancy that appears between its local information and the information obtained from the distant site. For this reason, we will define a new operator called parallel composition, that will be denoted by $||$.

Let $\mathcal{E}$ and $\mathcal{E}'$ be two event graphs. The **parallel composition** of $\mathcal{E}$ and $\mathcal{E}'$ is defined as $\mathcal{E}||\mathcal{E}' = \big(E \cup E', (\leq \cup \leq' \cup \leq_m)^*\big)$, where $\leq_m$ has the same meaning as before.

Proposition 4.1 *Let $\mathcal{E} = (E, \leq)$ be an event graph, e_s be an emission of a message, $e_r = m(e_s)$ be the corresponding reception, and $e = max_{l(e_r)}\{e \mid e \leq e_r \wedge e \neq e_r\}$ be the immediate predecessor of e_r on node $l(e_r)$. Then, we have:*

$$\downarrow_{\mathcal{E}}(e_r) = \downarrow_{\mathcal{E}}(e_s)||\big(\downarrow_{\mathcal{E}}(e) \circ (\{e_r\}, \emptyset)\big)$$

Proof:

Let us chose 3 events e_r, e_s, e with $m(e_s) = e_r$, and $e \prec_{l(e_r)} e_r$. Let us also note $\downarrow_r = \downarrow(e_r)$, $\downarrow_s = \downarrow(e_s)$ and $\downarrow_e = \downarrow(e) \circ (\{e_r\})$. Then we want to show that $\downarrow_r = \downarrow_s || \downarrow_e$. Let us show that the cover relation of the two partial orders is the same.

First, $\downarrow_e$ and $\downarrow_s$ are prefix of $\downarrow_r$ because $e \prec e_r$ and $e_s \prec e_r$. So $(\downarrow_s \cup \downarrow_e) \subset \downarrow_r$.

Second, by adding message causalities we ensure that the union graph $(\downarrow_s \cup \downarrow_e)$ is an event graph. So $\downarrow_r$ and $(\downarrow_s \,||\, \downarrow_e)$ have the same number of events and the causality relation of $\downarrow_r$ is included in causal order of $(\downarrow_s \,||\, \downarrow_e)$. As each one is an event graph, they are the same, ie. $\downarrow_r = \downarrow_s \,||\, \downarrow_e$ which is what we want. □

This property shows how to update local histories computed by each node upon reception of a messages when history is attached to messages. From these more accurate local views, nodes can then compute a local part of an abstract graph. We saw in last section how to split a centralized execution into singleton event graphs, how to abstract each of them, and then how to build the abstract graph of the initial event graph from these abstract pieces. In a distributed setting, each node maintains a local view of the same execution, and the abstraction must be built out of this set of local knowledges. This notion of local view of an execution is formally defined as a vertical cut. A **vertical cut** $h : \mathbb{B} \to \mathbb{B}^n$ is a function producting a n-uple of local execution on n nodes from a global computation. For each event graph $\mathcal{E} \in \mathbb{B}$, we have:

$$h(\mathcal{E}) = \big(\downarrow (max_1(\mathcal{E}), \ldots, \downarrow (max_n(\mathcal{E}))\big) = (h_1(\mathcal{E}), \ldots, h_n(\mathcal{E}))$$

Proposition 4.2 *Let us denote by* $\underset{i \in 1..k}{||} x_i = x_1 || \ldots |||| x_k$ *the parallel composition of event graphs* $x_1, \ldots x_k$. *We have* $||.h = Id_{\mathbb{B}}$, *i.e. for each event graph* $\mathcal{E} \in \mathbb{B}$: $\mathcal{E} = \underset{i \in 1..n}{||} h_i(\mathcal{E})$

Proof:

We want to show that $\mathcal{E} = ||h_i$. This theorem is the generalization of proposition 4.1. We show it in the same way as proposition 4.1, by saying that each h_i is a subset of $\mathcal{E}$ and then $\bigcup h_i \subset \mathcal{E}$. Then $||h_i$ is the smallest event graph closure of $\bigcup h_i$ with same number of event than $\mathcal{E}$, which means that $\mathcal{E} = ||h_i$. □

This property means we can decompose an execution into local histories with the h operator and then reconstruct it with the $||$ operator. This allows us to perform local computations on each node, relative to its history, because we know how to reconstruct a complete execution. This proposition is not sufficient to build a distributed abstraction framework: the size of an execution may grow infinitely, so it is not realistic to sent a complete history to a neighbor. Furthermore, there is redundancy in local views, and several nodes may build the same part of the abstract graph. Note also that an atom may appear in the execution, but not in any of the local views computed by each site. We will show in the rest of the paper how to exploit redundancy to send the minimal useful information, and how to make sure that all atoms are eventually detected.

We can define parallel composition operator for abstract graphs similarly to what has been done for event graphs. Let $\mathcal{A}_1 = (A_1, \alpha_1)$ and $\mathcal{A}_2 = (A_2, \alpha_2)$ be two abstract graphs. The **parallel composition** of abstract graphs $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined as: $\mathcal{A}_1 || \mathcal{A}_2 = \big(A_1 \cup A_2, (\alpha_1 \cup \alpha_2 \cup \alpha_m \cup \alpha_s)^*\big)/\mathcal{CC}$, where

- α_m is the same as already defined for sequential composition.

- $\alpha_s = \{(a, a'), (a', a) \mid a \in A, a' \in A', a \cap a' \neq \emptyset\}$ are set of causalities added to merge non disjoint atoms.

Proposition 4.3 *Parallel composition is distributive over abstraction:*

$$\mathcal{A}(\mathcal{E})||\mathcal{A}(\mathcal{E}') = \mathcal{A}(\mathcal{E}||\mathcal{E}')$$

Proof:
We want to show that $\mathcal{A}(\mathcal{E})||\mathcal{A}(\mathcal{E}') = \mathcal{A}(\mathcal{E}||\mathcal{E}')$. Let us denote $\mathcal{A}(\mathcal{E}||\mathcal{E}')$ by $(\mathcal{E}||\mathcal{E}')/\mathcal{R}$ and $\mathcal{A}(\mathcal{E})||\mathcal{A}(\mathcal{E}')$ to be $(\mathcal{E}||\mathcal{E}')/\mathcal{R}'$. We can proceed as in proposition 2.2 and show that $\mathcal{R} = \mathcal{R}'$, if we write $\alpha_p = \alpha_m \cup a_s$. □

So, property 4.1 on the past of events can be easily adapted to obtain the following property:

Proposition 4.4 *Let $\mathcal{E} = (E, \leq)$ be an event graph, e_s be an emission of a message, $e_r = m(e_s)$ be the corresponding reception, and $e = max_{l(e_r)}\{e \mid e \leq e_r \wedge e \neq e_r\}$ be the immediate predecessor of e_r on node $l(e_r)$. Then, we have:*

$$\mathcal{A}(\downarrow_{\mathcal{E}} (e_r)) = \mathcal{A}(\downarrow_{\mathcal{E}} (e_s))||\big(\mathcal{A}(\downarrow_{\mathcal{E}} (e)) \circ \mathcal{A}((\{e_r\}, \emptyset))\big)$$

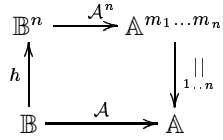
Proof:
We will use notation of proof of proposition 4.1.We want to show that $\mathcal{A}(\downarrow_r) = \mathcal{A}(\downarrow_s)||\mathcal{A}(\downarrow_e)$. This follows from proposition 4.1 and proposition 4.3. □

This proposition indicates how to compute incrementally the abstract graph $\mathcal{A}(\downarrow_{\mathcal{E}} (e_r))$ from local knowledge $\mathcal{A}(\downarrow_{\mathcal{E}} (max(e \mid l(e) = l(e_r)))$ and from $\mathcal{A}(\downarrow_{\mathcal{E}} (m^{-1}(e_r)))$ added on received messages. A node can receive only $\downarrow_{\mathcal{E}} (m^{-1}(e_r))$ and compute its abstraction before composing it with its local abstract history. We also have the following theorem on abstraction reconstruction from local histories:

Theorem 4.1 $\mathcal{A}(\mathcal{E}) = \underset{i \in 1..n}{||} \mathcal{A}(h_i(\mathcal{E}))$

Proof:
We want to show that $\mathcal{A}(\mathcal{E}) = ||\mathcal{A}(h_i)$. This follows from theorem 2.1 and proposition 4.3. □

This can be resumed to the following commutative diagram:



This theorem means that a correct abstraction can be obtained out of abstractions of local histories, however message size and local memory needed may grow infinitely as they may carry to much information. Let us formalize this more precisely:

4.2 Local and global coherence

Let us assume that a and b are two events localized on different sites $l(a)$ and $l(b)$, as shown in Figure 2. Let us also assume that a is an emission of a message m to node $l(b)$, and that b immediately precedes the reception of m corresponding to the emission a. We want to compute efficiently $\downarrow_{\mathcal{E}}(a) \cup \downarrow_{\mathcal{E}}(b)$, i.e. we want to know the information that must be carried by message m. Let us denote by $P_{ab} = \downarrow_{\mathcal{E}}(a) \cap \downarrow_{\mathcal{E}}(b)$ the intersection of a and b's pasts. Then we have $\downarrow_{\mathcal{E}}(a) \cup \downarrow_{\mathcal{E}}(b) = (\downarrow_{\mathcal{E}}(a) - P_{ab}) \uplus \downarrow_{\mathcal{E}}(b)$ where $\uplus$ is the disjoint union operation, and $\downarrow_{\mathcal{E}}(b)$ is the local knowledge of node $l(b)$ when action b is executed. Clearly, node $l(b)$ does not need the complete knowledge at node a to update its view of the execution. $(\downarrow_{\mathcal{E}}(a) - P_{ab})$ is the only part of a's past that is needed, and should hence be attached to message m. However the event set P_{ab} cannot be computed effectively, as according to the definition, it depends on a and b's past, and node $l(a)$ does not always know b's past when message m is sent. Hence we will over-approximate P_{ab} in order to send the smallest possible set of events containing P_{ab}.

Let $\mathcal{E} = (E, \leq)$ be an event graph. The **update suffix** of $\mathcal{E}$ on node j for node i is denoted $\mathcal{S}_{ji}(\mathcal{E}) = (S_{ji}, \leq_{ji})$ and is the biggest suffix of $\mathcal{E}$ closed by causal precedence that does not contain events on node i and message emissions to node i. Intuitively, an update suffix contains all the information that may not be known by node i. More formally, we have:

- $e \in S_{ji} \Rightarrow l(e) \neq i$ and $\uparrow_{\mathcal{E}}(e) \subset S_{ji}$
- $e \in S_{ji} \cap \mathbb{E}_s \Rightarrow l(m(e)) \neq i$

Algorithm 3 computes the update suffix $\mathcal{S}_{ji}$ of event graph $\mathcal{E} = (E, \leq)$ for node i, assuming that $\mathcal{E}$ is the history known by node j.

Algorithm 3 Update suffix computation on node j

update_suffix: $\mathbb{Z} \times \mathbb{B} \to \mathbb{B}$

```
1:  Input i, E = (E, ≤)
2:  S_ji = (∅, ∅)
3:  while E ≠ ∅ do
4:     Choose e ∈ max(E)
5:     E = E − {e}
6:     if (l(e) = i) OR (e ∈ E_s and l(m(e)) = i) then
7:        E = E − {e' | e' ≤ e}
8:     else
9:        S_ji = ({e}, ∅) ∘ S_ji
10:    end if
11: end while
12: Output: S_ji
```

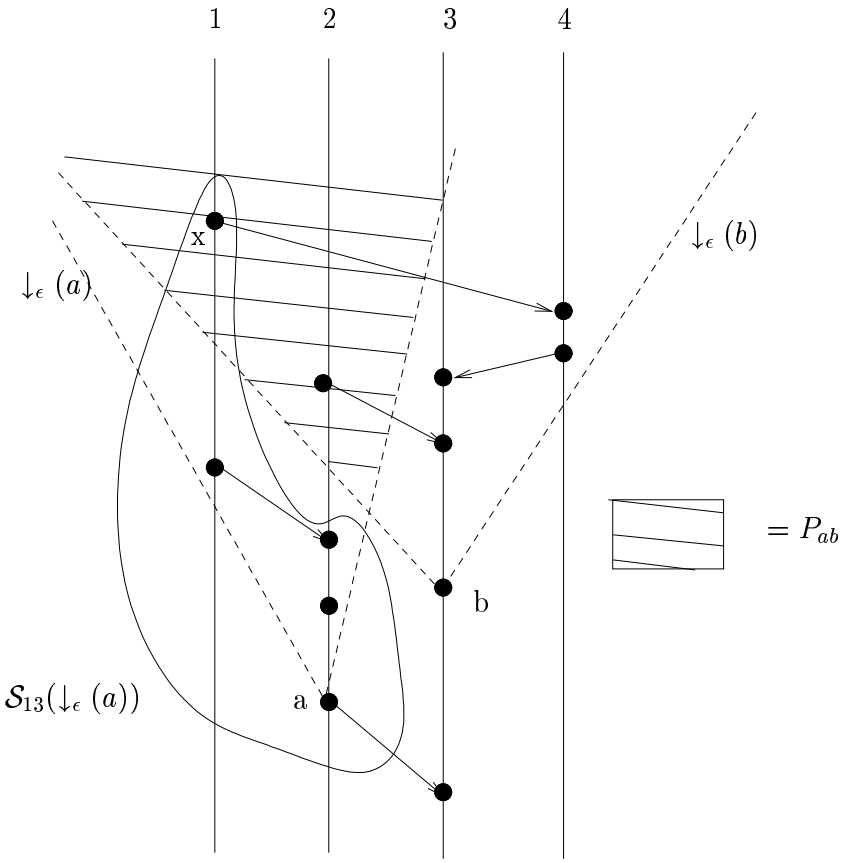


Figure 2: Updates suffixes

The complexity of this algorithm is in $O(|E|)$ (it is almost similar to a spanning tree exploration). Note that update suffixes must be computed from event graphs as abstract graphs do not contain enough information on causality. Moreover, in a network of n nodes, each node will maintain an update suffix per neighbor, and attach it to the messages it sends. Note also that even if update suffixes limit the size of messages sent, their size may still grow indefinitely. For example when a node k receives messages but never sends any acknowledgment to its neighbors, the update suffix associated to node k on each neighbor node will grow infinitely.

Figure 2 illustrates how to build this update suffix. This figure represents a distributed execution on four nodes $\{1, 2, 3, 4\}$. Node 2 has no way to know that event x is already contained in the causal past of b, and hence in P_{ab}. So, when event a (an emission of a message to node 3) is executed, the message sent is tagged with the update suffix $\mathcal{S}_{13}(\downarrow a)$, that contains the information needed by node 3 to reconstruct $\downarrow m(a)$.In figure 2, the pasts of a and b are represented by cones delimited by dotted lines. The common past of a and b is represented by the dashed intersection, and the update suffix is the set of events delimited by a plain line. Note that event x is an event of the update suffix that is already in the common past of a and b.

Proposition 4.5 *Let a be an event of an event graph $\mathcal{E}$ and b be the immediate predecessor of $m(a)$ on node $l(b)$. Then $(\downarrow_{\mathcal{E}} (a) - P_{ab}) \subset \mathcal{S}_{l(a)l(b)}(\downarrow_{\mathcal{E}} (a))$.*

Proof:

We want to show that $(\downarrow (a) - P_{ab}) \subset \mathcal{S}_{l(a)l(b)}(\downarrow (a))$. Let e be an event of $(\downarrow_{\mathcal{E}} (a) - P_{ab})$. First, $l(e) \neq l(b)$: if it is not the case, b and e are comparable, i.e. $e \leq b$. Then e is in P_{ab}, which is impossible. Second, if $e \in \mathbb{E}_s$, then $l(m(e)) \neq l(b)$ or $e = a$. If $l(m(e)) = l(b)$, it means that there exists e' such that $l(e') = l(b)$ and $e' = m(e)$. Then: $e \leq e' \leq b$, that means e is in P_{ab} which is impossible. Finally, $\downarrow (\downarrow (a) \cap \downarrow (b)) = (\downarrow (a) \cap \downarrow (b))$, so each event bigger than e is also in $(\downarrow (a) - P_{ab})$. Thus, $\uparrow (e) \subset S_i$. □

This proposition means that update suffixes $\mathcal{S}_{ij}$ contains the information needed by node j to update accurately its knowledge of running execution.

Proposition 4.6 *Let $\mathcal{E} = (E, \leq)$ be an event graph, e_s be an emission of a message, $e_r = m(e_s)$ be the corresponding reception, and $e = max_{l(e_r)}\{e \mid e \leq e_r \wedge e \neq e_r\}$ be the immediate predecessor of e_r on node $l(e_r)$. Then, we have:*

$$\downarrow_{\mathcal{E}} (e_r) = \mathcal{S}_{l(e_s)l(e_r)}(\downarrow_{\mathcal{E}} (e_s))||(\downarrow_{\mathcal{E}} (e) \circ (\{e_r\}, \emptyset))$$

Proof:

Consequence of propositions 4.1 and 4.5. □

These properties mean that update suffixes are sufficient to compute $\downarrow_{\mathcal{E}} (e_s) \cup \downarrow_{\mathcal{E}} (e_r)$. An update suffix sent to a node i by a node j represents all the past of j that i may still

not know. Sending an update suffix instead of a complete history is sufficient and reduces the size of data that is piggy-backed by messages.

Within our architecture, atoms must be sent to a supervisor when they are detected. The supervisor then builds an abstraction out of the received atoms. Note however that all nodes participating in the execution of a given atom a might signal it to the supervisor. To avoid redundancy, we impose that all atoms are signaled by a single node. As atoms are not known from the beginning of the execution, we designate the node with the lowest identification executing one of the last events in a (i.e $resp(a) = min(l(max(a)))$) as **responsible** for the notification to the upper level. Then the following property, that indicates that no atom is lost when a single responsible is designated holds trivially:

Proposition 4.7 *Let $\mathcal{E} \in \mathbb{E}$ be an event graph. Then,* $\bigcup_{i \in 1..n} resp^{-1}(i) = \{a \in \mathcal{A}(\mathcal{E})\}$.

Proof:

We know that $\bigcup resp^{-1}(i) \subset \{a \in \mathcal{A}\}$. Then, by cardinality, we can conclude that property4.7 holds. □

From the definition of update suffixes, and using propositions 4.6 and 4.7, we can immediately obtain the algorithm 4 that is executed by each node when an event e occurs. Note that even if the update suffix mechanism allows to reduce the memory used, this does not guarantee that algorithm 4 will eventually output an abstract graph, nor that the size of messages remains bounded. Indeed, executions do not always have the shape of concatenations of a finite set of finite atoms. Consider, for example, the execution of Figure 3. This execution can not be decomposed into communication closed subsets of events. Consequently, the size of the event set kept in memory by each process may grow infinitely. Furthermore, a node may never obtain enough information to deduce that an atom is stable. So, abstract graphs and update suffixes may still grow infinitely. Let us formalize sufficient conditions for which algorithm 4 produces a result and limits the size of information added to messages:

i) The execution does not have the shape of a braid as in Figure 3. Thus there is a K such that vertices of an abstract graph will always be subsets of less than K events: $\forall \mathcal{E} = (E, \leq),\ \forall e \in E,\ |\{e' \in E \mid e \mathcal{R}_{\mathcal{E}} e'\}| < K$.

ii) There exists a k such that if we chose a window W of size greater than k (i.e. a downward and upward closed subset of events (w.r.t $\leq$) with more than k events) in our execution all communication links of W are symmetric. More formally, we can write:

$$\forall W \text{ such that } \downarrow(W) \cap \uparrow(W) = W \quad \text{and } |W| > k, \big(e, m(e) \in W\big) \Rightarrow \big(\exists e', m(e') \in W \mid l(e) = l(m(e')) \wedge l(m(e)) = l(e')\big)$$

Figure 4 shows an example of execution where nodes $\{1, 2, 3\}$ can never forget a part of the execution they have observed, even if this execution could be decomposed offline

Algorithm 4 Behavior on a node $i \in \mathbb{N}$ at execution of an event e

Known data: i, $\mathcal{A}_i$, $\{\mathcal{S}_{ij}\}_{i \neq j}$

$\mathcal{A}_i = \mathcal{A}_i \circ (\{\{e\}\}, \emptyset)$
for all $j \neq i$ **do**
  $\mathcal{S}_{ij} = \mathcal{S}_{ij} \circ (\{e\}, \emptyset)$
end for

if e is reception of a message from node k with tag T
if $t(e) = r$ **then**
  $\mathcal{A}_i = \mathcal{A}_i || \mathcal{A}(T)$
  for all $j \neq i$ **do**
    $\mathcal{S}_{ij} = \mathcal{S}_{ij} || T$
  end for
end if

common part
for all $j \neq i$ **do** $\mathcal{S}_{ij} = \text{UpdateSuffix}(S_{ij})$ **end for**
for all $a \in \text{StablePrefix}(\mathcal{A})$ **do**
  $\mathcal{A}_i = \mathcal{A}_i - a$
  if $i = \min((l(a))$ **then** Send a to *supervisor* **end if**
end for

if e is emission of a message m to a node k
if $t(e) = s$ **then**
  Send m with tag $\mathcal{S}_{ik}$ to k
end if

into atoms. Indeed, process 2 never learns that the occurrences of atom a_2 have been detected. If process 2 and 3 never communicate, the set of atoms kept in memory by process 2 can hence grow infinitely.

Proposition 4.8 *If assumptions i) and ii) are always true during the execution, then for all nodes, the size of the local abstract graph $\mathcal{A}_i$ kept in memory by each node $i \in 1..n$ is bounded. Furthermore, the size of all update suffixes carried by messages is also bounded.*

Proof:

We want to show that we can bound size of piggy-backed data (ie. update suffixes really sent) and of local abstraction if bounded atom size *i*) and symmetric link *ii*) assumptions are verified.

First, let us suppose that we have an unbounded local abstract graph. It means that either an event class has an unbounded number of event (which is impossible if bounded

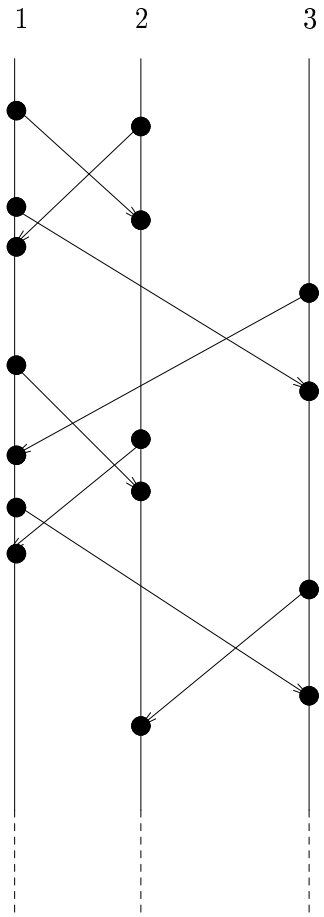


Figure 3: Infinite braids

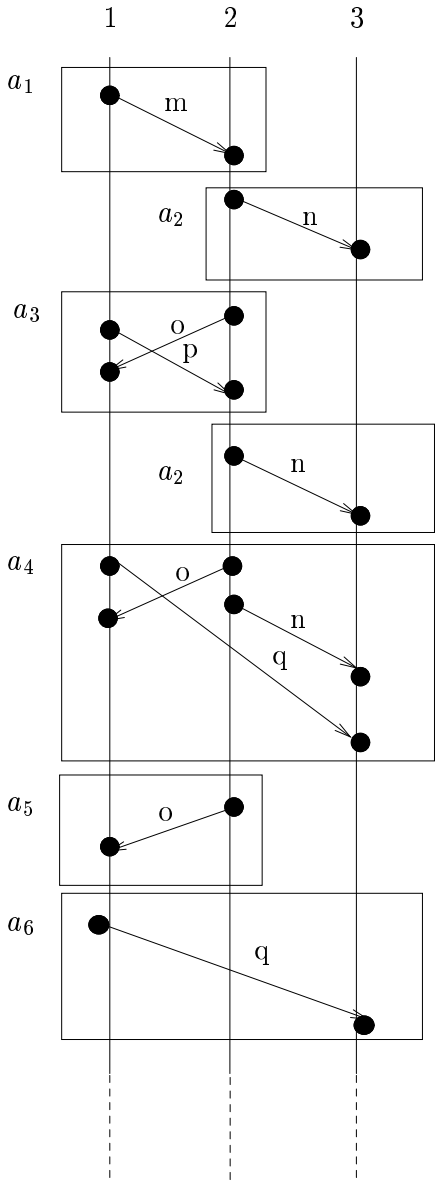


Figure 4: An execution that never synchronize

atom size assumption is verified) or this abstract graph has an unbounded number of vertices. That means we have an unbounded number of non event classes that are not communication closed: ie. an unbounded number of messages which are not received. If we have the symmetric link assumptions, we know that each site receiving a message will answer to the sender in a bounded amount of time. That means that each non paired emission event will be paired in a bounded amount of time. As we also assume that event class size is bounded, thus we have size of local abstract graph bounded.

Second, let us have an update suffix sent from node i to node j. Then, if we have symmetric link assumption, we already sent in a bounded amount of time in the past a message from node i to j: thus current update suffix size is also bounded.

Finally, we can say nothing about the size of update suffixes that are not sent. Their size may grow infinitely if the execution consists of two sets of nodes that never communicate. □

The two assumptions above are easily met by any algorithm that forces a synchronization from time to time among its components. The solution proposed is architecture-friendly, but it can slow atom detection, due to the loss of some transitivity relations. Note also that if during the execution two groups of processes evolve independently without ever exchanging messages, then the size of $\mathcal{S}_{ij}$'s that each group computes for the other may still grow infinitely. A quick optimization could be to send control messages when the suffix size becomes too big. Algorithm 4 detects atoms of a distributed execution on the fly. Once an atom is detected, it is sent to a supervisor, that maintains an abstract view of the ongoing computation. Figure 1 depicts the architecture of this system and the information exchanged among nodes and to the supervisor. Note however that as we do not make any assumption on the type of communication channel between nodes and their supervisor, atoms are not necessarily received in their order of appearance. This problem can be solved if nodes maintain a vectorial clock of atoms and tag the freshly detected atoms with it, in the usual way proposed by Fidge & Mattern [2, 8](however we will not detail this additional mechanism it in this paper).

4.3 Bounds

As already discussed, the algorithm 4 does not alway run with finite memory. A possible way to limit the memory needed on each node is to forget the parts of a node's past that are not essential to compute the next atoms.

Proposition 4.9 *Let e_s be an emission of a message, $e_r = m(e_s)$ be the corresponding reception, and e be the immediate predecessor of e_r on the same node $j = l(e_r)$. Let us denote by $K_i(\mathcal{E}) = \mathcal{E} - \{a \in StablePrefix(\mathcal{E}) | i \notin l(a)\}$ and $K(\mathcal{E}) = \mathcal{E} - \{a \in StablePrefix(\mathcal{E})\}$ the unstable part of $\mathcal{E}$. Then we have:*

$$K_j(\downarrow e_r) \subseteq K(S_{ij}(\downarrow e_s)) || K_j(\downarrow e \circ \{\{e_r\}, \emptyset\})$$

Proof:
First, we have $K(S_{ij}(\downarrow_s)) = K_j(S_{ij}(\downarrow_s))$ as no event localized on j belongs to S_{ij}. Second, as we have $\downarrow_r = S_{ij}(\downarrow_s)||\downarrow_e$ (proposition 4.6), we have also $K_j(\downarrow_r) \subset K(Sij(\downarrow_s))||K_j(\downarrow_e)$. □

This proposition means that from $K(\mathcal{S}_{ij}(\downarrow_{\mathcal{E}}(e_s)))$ received by node j and a local view $K_j(\downarrow e \circ \{\{e_r\}, \emptyset\})$, node j can reconstruct a local view that contains $K_j(\downarrow e_r)$, i.e. a local view that contains stable atoms where j is involved.

Theorem 4.2 *For all* $\mathcal{E} \in \mathbb{B}$ *we have:* $\underset{i\in 1..n}{||} K_i(h_i(\mathcal{E})) = \mathcal{E}$ *and* $\underset{i\in 1..n}{||} \mathcal{A}(K_i(h_i(\mathcal{E}))) = \mathcal{A}(\mathcal{E}))$

Proof:
From proposition 4.2 we have $||h_i = \mathcal{E}$. First, K_i does not add events or causality. So $||K_i(h_i) \subset \mathcal{E}$. Second, each event of $\mathcal{E}$ also belongs to $K_{l(e)}(h_{l(e)})$ because K_i does not remove events located on i. Then each relation $e \leq_m e'$ in $\mathcal{E}$ also belongs to $K_{l(e)}$ because e and e' are in the same atom a and $l(e) \in l(a)$. And for relation $e \leq_l e'$ also belongs to $K_{l(e)}$ because e and e' take place on $l(e)$. Thus $\mathcal{E} \subset ||K_i(h_i)$. So, we can conclude that $||K_i(h_i) = \mathcal{E}$. □

This property means that from local histories computed on each node, if we do not take into account atoms involving only other nodes, then a global execution can still be computed. This does not mean however that all atoms of an execution always appear in one of the local histories computed by nodes. An atom may remain incomplete in all local histories, and be rebuilt by parallel composition of histories (we can sometimes find an atom a in the stable prefix of $\mathcal{A}(\mathcal{E})$ such that $\forall i \in 1..n, a \notin \mathcal{A}\big(K_i\big)\big)$. However, if the network architecture guarantees good properties of executions, it is still possible to forget a part of the past.

Proposition 4.10 *Let* $\mathcal{E}$ *and* $\mathcal{E}'$ *be two disjoint abstract graphs, and let* a *be an atom of the stable prefix of* $\mathcal{A}(\mathcal{E})$. *If* $\forall i \in l\big(max(a)\big)$, $\exists e, m(e)$ *such that* $i = l(e)$ *and* $resp(a) = l\big(m(e)\big)$, *then:* $a \in \mathcal{A}\big(K_{resp(a)}(\downarrow max_{resp(a)}(\mathcal{E} \circ \mathcal{E}'))\big)$

Proof:
We want to show that $a \in \mathcal{A}(\mathcal{E}) \Rightarrow a \in \mathcal{A}(K_{resp(a)}(\downarrow max_{resp(a)}(\mathcal{E} \circ \mathcal{E}')$ if nodes in $l(a)$ are causally related to $resp(a)$ in of $\mathcal{E}'$. If for each i in $l(a)$ we have a message chain terminating to $resp(a)$ in $\mathcal{E}'$, then we never remove events of a in update suffixes, and a complete message arrives to $resp(a)$. Thus, all events of a are communicated to $resp(a)$ and it can send a to the supervisor. □

This proposition means that after execution of any atom a, if all nodes terminating the atom's execution communicate with $resp(a)$, then nodes can always forget stable atoms in which they do not participate while preserving correctness of atoms detection. Note that there is no bound on the size of $\mathcal{E}'$. So, in general, the supervisor can be late by a certain (or even an unbounded) number of atoms. When it is sure that after execution of any atom

a, all participating instances will eventually communicate with the node that is responsible for a, then theorem 4.2 is eventually satisfied by all executions, and the common part of the algorithm can be replaced by the algorithm 5 below. These requirements are easily met by architectures that impose nodes to communicate with their neighborhood from time to time, or to terminate their executions with a broadcast. This enhancement allows us to save memory by forgetting all atoms that do not concern a node. When an event graph $\mathcal{E}$ is such that $\forall a \in \mathcal{A}(\mathcal{E})$, $(\mathcal{E} - \uparrow_{\mathcal{E}} (a))$, $\uparrow_{\mathcal{E}} (a)$ and a satisfy property 4.10, we will say that atoms of $\mathcal{E}$ are **recognized locally**.

Theorem 4.3 *If an eventually infinite execution $\mathcal{E}$ verifies assumptions i) and ii),and that its atoms are recognized locally, then the size of each $\mathcal{S}_{ij}, i, j \in 1..n$ and each locally computed abstract graphs $\mathcal{A}_i, i \in 1..n$ is bounded.*

Proof:

We want to show that all the data structures used by the algorithm are of bounded size when the 3 following assumptions hold: bounded atom size, symmetric communication links, and future communication with the responsible in a bounded amount of time.

We already shown in proposition 4.8 that local abstract graphs and update suffixes sent are bounded. We need to show that update suffixes keep in memory and never sent are also bounded. As we now keep $K(S_{ij})$ (and no more all S_{ij}'s), we remove atoms that will never be useful for the neighborhood: in on a node i, we remove all atoms that only concern a set of nodes that evolve independently. Thus, we ensure to always keep a bounded number of events in memory and we ensure that each atom is sent to supervisor in a bounded amount of time. □

Theorem 4.3 provides sufficient conditions for which the detection algorithm refined with the enhancement of algorithm 5 provides a correct abstraction and works with finite memory. Furthermore, when these conditions apply, we can be sure that an atom is sent to the supervisor within a bounded number of execution steps after its appearance in the execution.

Algorithm 5 Common part enhancement to save memory

```
for all  j ≠ i   do  S_ij = UpdateSuffix(S_ij) end for
for all a ∈StablePrefix(A) do
  A = A − a
  for all j ≠ i  do
    if a ⊂ S_ij then  S_ij = S_ij − a end if
  end for
  if  i = min((l(a)) then  Send a to supervisor) end if
end for
```

5 Conclusion

This paper has proposed an online algorithm to build an abstract view of an execution. A running execution is partitioned into atomic communication patterns. Each node in the system computes its own decomposition from his knowledge of the execution, and forwards the minimal useful information to its neighbors. This information can be piggy-backed on messages of the execution. Note that the memory needed by each node and the size of the data sent remains bounded when execution can synchronize from time to time, and when it does not have the shape of an infinite braid. The algorithm proposed implements an online distributed abstraction mechanism, which obviously does not achieve an optimal compression rate. All nodes in the system compute an alphabet of atoms and partial order over occurrences of these atoms, that are collected by a supervisor. A better compression can be obtained by the supervisor using the abstract alphabet of atoms with the principles given in [11]: either chose a peculiar linearization of the abstract order and compress it, or compress the unique string obtained from the Cartier-Foata normal form of the abstract order. However, the initial abstraction has the nice property that any event graph obtained for a stable prefix of an abstract graph is a distributed control point. Furthermore, the abstraction is unique (up to isomorphism).

References

[1] R. Alur, S. Chaudhuri, K. Etessami, S. Guha, and M. Yannakakis. Compression of partially ordered strings. In R. Amadio and D. Lugiez, editors, *Proc. of CONCUR 2003*, number 2761 in LNCS, pages 42–56. Springer Verlag, 2003.

[2] C.J Fidge. Partial orders for parallel debugging. In *Proc. of the ACM SIGPLAN/SIGOPS International Workshop on Parallel and Distributed Debugging*, pages 183–194, 1989.

[3] A. Goel, A. Roychoudhury, and T. Mitra. Compactly representing parallel program executions. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 191–202, 2003.

[4] J.G. Henriksen, M. Mukund, N. Kumar, and P.S. Thiagarajan. On message sequence graphs and finitely generated regular msc languages. In *Proc. of ICALP 2000*, number 1853 in LNCS, pages 675–686. Springer Verlag, 2000.

[5] L. Hélouët and P. Le Maigat. Decomposition of message sequence charts. In *Proc. of SAM'2000, 2nd conference on SDL and MSC*, 2000.

[6] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, September 1999.

[7] J. Larus. Whole program paths. In *Proc. of the ACM Conference on Programming Languages Design and Implementation*, pages 259–269, 1999.

[8] F. Mattern. Virtual time and global states of distributed systems. In *proc. of the ACM SIGPLAN/SIGOPS International Workshop on Parallel and distributed debugging*, pages 215–226, 1989.

[9] Netzer and Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEETPDS: IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

[10] C. Nevill-Manning and Witten I.H. Identifying hierarchical structure in sequences: a linear time algorithm. *Journal of Artificial Intelligence Research*, (7):67–82, sep. 1997.

[11] S. Savari. On compressing interchange classes of events in a concurrent system. In *Proc. of the Data Compression Conference (DCC'03)*, pages 153–162, 2003.

[12] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2), 1972.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399