RECHERCHE

Observation d'exécutions réparties : détection de propriétés d'ordonnancement

Claude Jard

IRISA/CNRS
Campus de Beaulieu
35042 Rennes Cedex
jard@irisa.fr

RÉSUMÉ. *L'interopérabilité des implantations de protocoles dans les réseaux de télécommunications, ou les systèmes répartis, est un objectif difficile à garantir. C'est pourquoi il est intéressant de pouvoir observer le fonctionnement du système et d'effectuer un diagnostic en ligne. Considérant une implantation répartie, et une propriété globale définissant un comportement attendu du système, le problème est de concevoir des testeurs capables de décider du respect ou non de cette propriété. Nous montrons l'importance de la capture les dépendances causales se produisant entre les actions observables du système, et proposons des méthodes générales, locales ou globales, pour détecter au vol la violation de propriétés globales d'exécutions réparties.*

ABSTRACT. *Testing is one of the major problem on developing communication software. It becomes a very challenging task when considering "interoperability testing". In that context, one must decide the correctness of distributed behaviours : given a global property involving different protocol entities, the problem is to design testers able to decide whether the system conforms or not with the property. We show the interest to track the causal dependencies occuring between the observable actions of the system, and propose some general methods to detect on-the-fly the violation of global properties of distributed executions.*

MOTS-CLÉS : *Réseaux, Protocoles, Systèmes répartis, Diagnostic, Observation, Causalité, Vérification.*

KEYWORDS : *Networks, Protocols, Distributed systems, Diagnostic, Observation, Causality, Verification.*

1. Introduction

La répartition de calculs sur des réseaux de processeurs pose fondamentalement la question de l'inter-opérabilité des équipements et des logiciels. Il y a au moins trois raisons suffisantes pour que le bon fonctionnement des implantations réparties ne soit pas toujours garanti, malgré même un développement particulièrement soigné et formalisé.

- l'effort de formalisation est souvent incomplet : des choix importants, influençant la correction du système, peuvent être laissés à l'implanteur.
- l'effort de vérification lui aussi, peut ne pas être complet : le résultat dépend de différentes hypothèses faites sur le comportement de l'environnement du logiciel (au niveau de l'utilisateur, aussi bien qu'au niveau de la communication) ; hypothèses qui pourraient ne pas être remplies à l'exécution.
- le test de conformité de chaque module de l'implantation ne prouve pas complètement la conformité. Des implantations non conformes peuvent passer positivement quand même le test.

C'est pourquoi observer en cours d'exécution le comportement des implantations réparties reste un moyen important pour sécuriser le fonctionnement des protocoles [RAF 91, JAR 94a]. Nous allons nous limiter à l'observation de propriétés sur l'ordonnancement des actions observables du système. Ces propriétés forment une classe significative de propriétés globales habituellement attendues dans les systèmes répartis, et constituent une base à des questions plus complexes portant sur des contraintes temps réel ou des probabilités.

Le concept de base pour parler de l'ordre des événements dans un système réparti est la notion de dépendance causale, introduite par Lamport en 1978. Nous montrerons dans cet article comment les dépendances causales entre un ensemble quelconque d'événements observables peuvent être capturées au vol, pour former ce que l'on appelle l'ordre de causalité.

La capture des dépendances causales n'est qu'une étape vers la vérification de propriétés. On montre comment à partir de la connaissance de l'ordre de causalité, on peut vérifier des propriétés comportementales (ordonnancement d'événements).

Nous avons distingué deux grandes approches :

- l'approche, dite *globale*, dans laquelle les événements du système observé sont extraits (avec une étiquette et une estampille), puis traités à l'extérieur. Cette approche est illustrée dans la figure 1(a) où l'on voit l'activité de trois processus parallèles P1, P2 et P3, provoquer l'envoi d'information de traces vers un observateur central, appelé Obs, et chargé d'émettre un diagnostic.
- l'approche, dite *locale*, qui restreint le type de propriétés que l'on peut vérifier aux propriétés ne portant que sur le passé causal d'un événement (i.e. ce qui est observable localement). L'avantage est que la vérification peut alors être effectuée localement sur le processus qui produit l'événement. Ceci permet une vérification elle-même répartie, dans laquelle les différents observateurs se coordonnent. Ceci est illustré dans la figure 1(b) où l'on voit des observateurs O1, O2 et O3, greffés sur chaque processus du système, et capables d'émettre un diagnostic en se coordonnant éventuellement à travers le réseau d'interconnexion des processus.

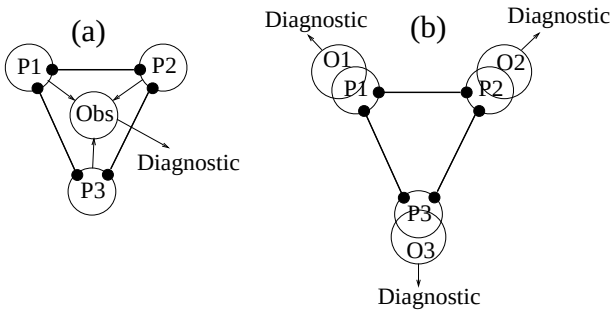

Figure 1. *L'observation globale (a) ou locale (b) d'un système formé de trois processus P1, P2 et P3*

Le reste de l'article est organisé comme suit : le deuxième paragraphe présente un petit exemple de protocole et de propriété. Il sert à illustrer le problème du diagnostic en environnement réparti et sera utilisé plus tard pour fournir un exemple de vérification globale et locale. Le paragraphe suivant présente le modèle formel utilisé et une méthode générale pour observer l'ordre de causalité. Les deux derniers paragraphes sont consacrés respectivement à l'exposé d'un algorithme de vérification globale, mis en œuvre sur un testeur centralisé, puis à l'exposé d'un algorithme distribué de vérification locale. Dans ce dernier cas, le test est effectué de manière répartie en associant un testeur à chaque implantation sous test.

2. Exemple de protocole

2.1. *Architecture et service*

Considérons l'architecture décrite par la figure 2(a) pour l'implantation sous test. Elle est formée de deux entités de protocoles (EP1 et EP2). Ces entités sont activées par des testeurs (Test1 et Test2) pilotant les primitives de service du protocole (ici, les messages *Req*, *Cnf* et *Ind*). Elles communiquent à travers les couches inférieures de l'architecture.

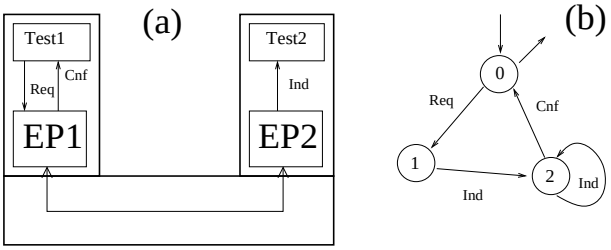

Figure 2. *(a): architecture de test; (b): automate spécifiant le séquencement* Req-Ind-Cnf *typique d'un service de transfert de données*

Imaginons maintenant que EP1 et EP2 implantent un protocole de transfert de données. L'émetteur (EP1) peut entamer un transfert en invoquant une requête (*Req*). Du point de vue de l'utilisateur, cette invocation aura pour conséquence l'occurrence d'une indication (*Ind*) chez le récepteur (EP2). Après la délivrance de cette indication, le protocole informera le service de l'émetteur par une confirmation (*Cnf*). Ce qui permettra à l'utilisateur de l'émetteur (Test1) de demander un nouveau transfert.

La propriété (globale) de séquencement attendu (*Req*→ *Ind*→*Cnf*) est modélisée dans la figure 2(b) par un automate d'état fini (0 est l'état initial et l'unique état accepteur). Ici, on a supposé de plus que l'on autorisait éventuellement plusieurs indications avant la confirmation.

2.2. *Exemple de situation d'erreur*

Considérons que les entités de protocole EP1 et EP2 sont équipées d'un protocole du type du "bit alternant". Rappelons le principe de ce protocole : il fait dialoguer un émetteur (EP1) et un récepteur (EP2) à travers une couche de communication non fiable. Pour se prémunir des pertes de messages, la donnée émise par EP1 est retransmise sur échéance de temporisation, jusqu'à la réception d'un accusé de réception envoyé par le récepteur EP2. Ce système provoquant des duplications lorsqu'une donnée non perdue est réémise, les données et les accusés sont munis d'un bit de séquence. Ce bit de séquence est géré par les entités pour détecter les doublons de données et d'accusés.

Plus précisément, voici un pseudo-code pour les entités. Le comportement est défini par un ensemble de transitions (condition : action) :

EP1 : **initialement** *état=repos; bit=0*
  trans *(état=repos)* **et** *réception(Req) :*
    émettre(donnée,bit); armer-tempo; état:=attente
  trans *(état=attente)* **et** *réception(accusé,b)* **et** *(b≠bit) :*
    réémettre(donnée,bit); réarmer-tempo
  trans *(état=attente)* **et** *réception(accusé,b)* **et** *(b=bit) :*
    alterner(bit); émettre(Cnf); état:=repos
  trans *(état=attente)* **et** *échéance-tempo :*
    réémettre(donnée,bit); réarmer-tempo

EP2 : **initialement** *bit=0*
  trans *réception(donnée,b)* **et** *(b=bit) :*
    émettre(Ind); émettre(accusé,b); alterner(bit)
  trans *réception(donnée,b)* **et** *(b≠bit) :*
    émettre(accusé,b)

Le protocole du bit alternant est conçu pour résister aux pertes de messages dans un milieu de communication qui ne peut pas déséquencer les messages (discipline premier-entré-premier-sorti entre les couples de processus). Imaginons cependant qu'un déséquencement puisse se produire. La numérotation modulo deux des messages qu'assure le protocole devient clairement insuffisante et des violations de la propriété de service peuvent apparaître. C'est ce que l'on voit dans le chronogramme de la figure 3, où la troisième indication se s'intercale pas correctement entre la troisième requête et la troisième confirmation.

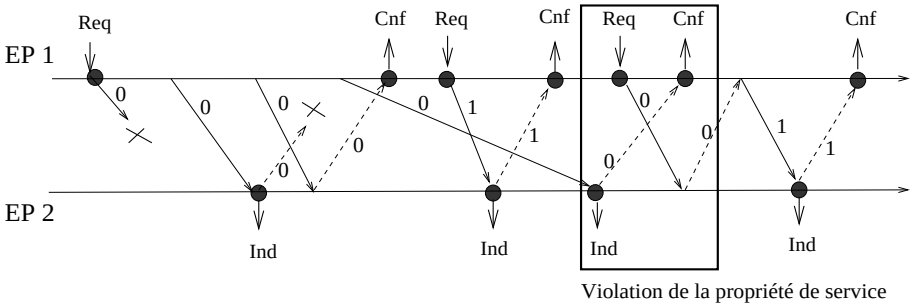

Figure 3. *Exemple d'exécution erronée : les événements observables de service sont les ronds gris sur la ligne de vie des processus. Les échanges de messages sont représentés par des flèches obliques (les données en traits pleins et les accusés en traits pointillés). La valeur du bit de séquence porté par les messages est aussi indiquée. La propriété globale de service est violée dans la zone rectangulaire.*

Toute la difficulté du problème réside dans l'automatisation du procédé de détection, étant bien entendu que l'ordonnancement pertinent ici ne repose pas sur une datation physique des événements mais sur les dépendances causales créées par les échanges de messages : dans notre exemple, la propriété est violée quelle que soit la façon de dessiner le chronogramme.

Le paragraphe suivant définit précisément la notion d'ordre entre les événements d'un calcul réparti et montre comment il peut être connu d'un observateur pour l'émission d'un diagnostic.

3. Modèle des exécutions réparties

3.1. *Causalité définie par les messages*

Nous considérons qu'une exécution répartie met en jeu un ensemble de processus $P_1, P_2, \cdots, P_n$. Les processus communiquent entre eux par l'envoi asynchrone de messages.

Chaque processus est assimilé à une activité séquentielle, et modélisé par l'exécution séquentielle d'un ensemble d'*événements*. La définition d'un événement dépend du niveau d'abstraction auquel se place le testeur. A un niveau d'abstraction donné, chaque événement est supposé *atomique*. Nous distinguons deux types d'événements :

- les événements *d'interactions*, c'est à dire l'envoi et la réception de messages,
- les événements *internes* des processus, c'est à dire tous les autres événements.

Nous appelons E_i l'ensemble des événements (internes et d'interactions) exécutés par le processus P_i, et E l'ensemble des événements formant l'exécution, c'est à dire $E = E_1 \cup E_2 \cup \cdots \cup E_n$.

En l'absence d'un référentiel de temps commun à l'ensemble des processus, il n'est pas possible d'ordonner totalement de façon fiable les événements de E. Il existe cependant une façon d'ordonner, mais partiellement, les événements entre eux. C'est la relation *happened before* introduite par Lamport[LAM 78] en 1978. Elle

est généralement appelée *précédence causale* ou *causalité*, bien qu'il ne s'agisse pas nécessairement de relation de cause à effet entre les événements, mais plutôt de relation nécessaire de précédence temporelle.

On distingue deux types de dépendances : d'abord, un processus étant un programme séquentiel, on peut ordonner totalement les événements de chaque processus entre eux, grâce au temps "local" du processus (nous notons $\leq_i$ l'ordre total sur l'ensemble E_i des événements du processus P_i). Ensuite, l'émission d'un message a toujours lieu avant la réception de celui-ci. La relation de causalité entre les événements observables est la fermeture transitive des deux relations que l'on vient de citer. Il s'agit d'une relation d'ordre, que nous noterons $\preceq$.

L'ensemble E des événements d'une exécution est partiellement ordonné par la relation d'ordre $\preceq$ suivante :

$$\forall e_i \in E_i, e_j \in E_j, e_i \preceq e_j \Leftrightarrow \begin{cases} i = j \text{ et } e_i \leq_i e_j \\ \text{(les deux événements ont lieu sur le même} \\ \text{processus et } e_i \text{ est exécuté avant } e_j) \\ \qquad \text{ou} \\ e_i \text{ est l'émission d'un message et } e_j \text{ sa réception} \\ \qquad \text{ou} \\ \exists e_k \text{ tel que } e_i \preceq e_k \preceq e_j \\ \text{(fermeture transitive)} \end{cases}$$

Pour raisonner formellement sur une trace d'exécution répartie et vérifier des propriétés, il est important de bien identifier la notion de niveau d'observation. Celui-ci est défini par l'ensemble des *événements observables*. En pratique, il s'agit des événements concernés par les propriétés à observer, c'est-à-dire généralement l'occurrence des différentes primitives de service du protocole. Nous appelons O_i le sous-ensemble des événements de P_i que l'on observe, et $O \subseteq E$ l'ensemble de tous les événements observables de l'exécution. La relation de causalité sur l'ensemble E induit une relation d'ordre sur l'ensemble O. Nous notons $\widetilde{O}$ cet ensemble ordonné.

Lorsque l'on décrit des propriétés sur les dépendances entre événements observables, on ne désigne pas directement leurs occurrences, mais plutôt leur type (ou nom). Dans notre exemple, ces événements ont pour noms *Req*, *Ind* et *Cnf*. C'est pourquoi on considère un alphabet Σ de noms servant pour spécifier des propriétés et une fonction de nommage λ associant un nom à chaque événement observable. Le triplet $(\widetilde{O}, \Sigma, \lambda)$ constitue donc le modèle formel d'une exécution répartie. On représentera de façon canonique une exécution par un graphe orienté (dit *de réduction transitive* ou encore *diagramme de Hasse*) dont les chemins forment les couples de la relation d'ordre. La figure 4 montre le graphe de l'ordre associé à notre exemple d'exécution de la figure 3.

3.2. *Processus observateur et états de l'exécution*

Comme il a été dit dans l'introduction, l'analyse peut être menée, soit de façon centralisée, soit de façon répartie. Si elle est répartie, on a alors une analyse effectuée localement sur chaque processus du système. Un observateur est superposé à chaque processus. Cet observateur est informé à chaque fois que le processus exécute un événement observable. Cette approche permet une réaction immédiate de

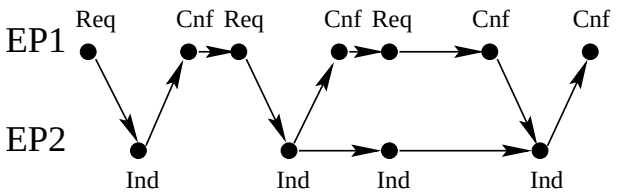

Figure 4. *Représentation graphique de l'ordre de causalité entre les événements observables de notre exemple*

l'observateur, mais le prive d'une vue globale du système (et donc restreint sa capacité d'analyse).

Le cas de l'analyse centralisée, lui, signifie qu'il existe un processus observateur qui effectue le diagnostic, mais ne prend pas part au calcul. Chaque processus du calcul envoie un message à cet observateur lorsqu'il exécute un événement observable. L'observateur peut ainsi obtenir une vue globale de l'exécution. Cependant, le temps d'acheminement des messages implique un délai entre l'exécution des événements observés et leur prise en compte par le processus observateur. De plus, l'observateur peut recevoir ces messages dans un ordre arbitraire, mais l'information de causalité que l'on va associer aux événements observables permettra à l'observateur de réordonner les événements dans un ordre compatible avec l'ordre de causalité.

La vision globale du calcul que peut acquérir l'observateur définit une notion d'*état* de l'exécution. Il s'agit tout simplement d'un certain degré d'avancement du calcul, c'est-à-dire d'un ensemble d'événements déjà exécutés par les processus. Tous les ensembles d'événements ne correspondent pas à des états : pour qu'un événement soit exécuté, il est nécessaire que tous les événements le précédant causalement dans le système aient été exécutés.

Un ensemble d'événements observables $I \subseteq O$ définit un état de l'exécution s'il est possible qu'à un moment donné, l'ensemble des événements exécutés soit exactement I. C'est à dire : $\forall o \in I, \ , \forall o' \in O, \ o' \preceq o \Rightarrow o' \in I$.

Lorsque l'on parlera d'*état local* d'un processus d'un i, on entend la restriction d'un état global aux événements exécutés par le processus i et à leurs prédécesseurs causaux.

Une exécution consiste donc à passer par une suite d'états; on commence par l'état vide (rien n'a été exécuté), puis on exécute un à un les événements, dans n'importe quel ordre pourvu que l'on respecte les relations de causalité. On finit par l'état formé de l'ensemble des événements de l'exécution. Il se trouve que l'ensemble de tous les états d'une exécution peut être ordonné par inclusion d'ensembles. La structure obtenue est un treillis, le *treillis des états (globaux)* du système. Chaque chemin dans ce treillis, allant du plus petit élément au plus grand élément, définit une exécution "possible", et chaque exécution possible définit un tel chemin.

L'absence de temps global nous empêche de savoir par quelle suite d'états l'exécution est "réellement" passée. Nous sommes donc en règle générale obligés de consid-

érer tous les états comme possibles, et une exécution répartie ne sera pas modélisé par un chemin dans le treillis mais par l'ensemble des chemins du treillis.

Nous désignons par $\widetilde{I(O)} = (I(O), \subseteq)$ le treillis des états de l'exécution $\widetilde{O}$, où $I(O)$ désigne l'ensemble des états. Comme la différence (ensembliste) de deux états successifs est d'exactement un événement, on peut représenter le treillis par le graphe de sa réduction transitive, les événements étiquetant cette fois-ci les arcs et non les nœuds. La figure 5 montre la structure du treillis de notre exemple d'exécution.

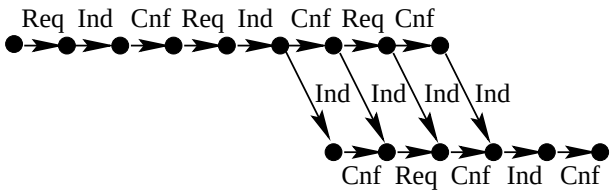

Figure 5. *Représentation graphique de l'ordre possible de succession des états de notre exemple*

3.3. *Calcul au vol de la causalité*

La première idée pour capturer efficacement la relation de dépendance causale est d'utiliser la décomposition induite par les processus. Considérons l'application suivante Δ qui associe à chaque événement observable, le nombre des événements qui le précèdent sur chacun des processus :

$$\begin{array}{rcll} \Delta & : & O & \longrightarrow & I\!N^n \\ & & x & \longmapsto & \mid \downarrow x \cap O_i \mid_{i \in [1 \ldots n]} \end{array}$$

où $\downarrow x = \{y \in O \ : \ y \preceq x\}$ désigne l'ensemble des événements prédécesseurs de x.

Le calcul des vecteurs (estampilles) permet de connaître exactement les dépendances causales entre événements observables d'après la propriété de codage suivante :

$$\forall x, y \in O, x \preceq y \Leftrightarrow \forall j \in [1..n], \Delta(x)[j] \leq \Delta(y)[j]$$

L'intérêt de ce codage est qu'il peut être calculé au vol pendant le déroulement de l'exécution (voir la figure 6 pour une illustration), sans ajout de messages de contrôle. Le principe est d'accumuler l'information vectorielle en utilisant les messages du calcul, messages qui justement induisent les dépendances causales.

L'algorithme suivant (dit des "estampilles vectorielles" de Fidge et Mattern [FID 88, MAT 89]), est installé sur chaque processus P_i :

Données:
  $\Delta_i[1..n]$ *tableau d'entiers;*
Initialisation:
  $\forall j \in 1..n : \Delta_i[j] := 0;$
Quand P_i *produit un événement observable* x*:*
  $\Delta_i[i] := \Delta_i[i] + 1;$
Quand P_i *envoit un message à* P_k*:*

$\Delta_i[1..n]$ *est passée dans le message;*
Quand P_i *reçoit de* P_k *un message portant* Δ_k*:*
$\forall j \in 1..n$: **faire**
si $\Delta_i[j] < \Delta_k[j]$ **alors** $\Delta_i[j] := \Delta_k[j]$;

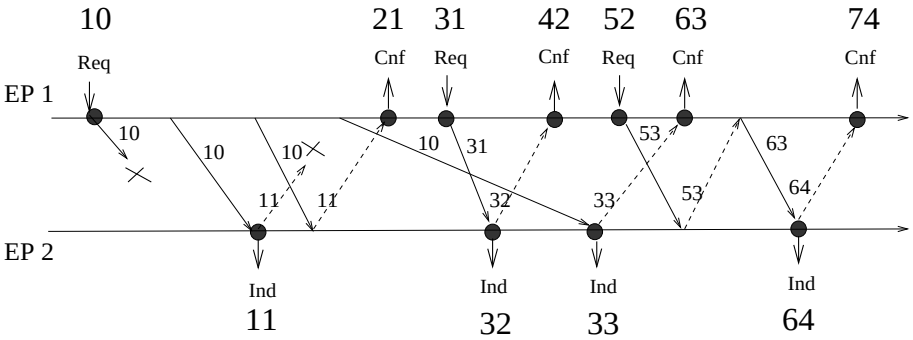

Figure 6. *Estampilles vectorielles : elles sont associées à chaque événement observable, et représentées par un couple d'entiers. Les valeurs transmises dans les messages sont aussi figurées*

Cet algorithme possède plusieurs variantes pour essayer de diminuer la quantité d'information à transmettre dans les messages du programme sous test. Une première classe de solutions consiste à observer une information moins précise, mais suffisante pour permettre la reconstruction de la relation de dépendance [FOW 90, BAL 92, JAR 96]. Une seconde approche consiste à exploiter les caractéristiques du réseau de processeurs (topologie, structure des canaux de communication, ...)[SIN 92]).

4. Approche de détection globale

4.1. *Assurer une observation causale*

L'approche globale suppose que les événements observables sont transmis à un observateur unique, chaque événement portant une estampille permettant de le comparer aux événements déjà reçus. Bien entendu, l'ordre "physique" d'arrivée des messages à l'observateur peut se trouver en contradiction avec l'ordre causal (un événement e précédant causalement un événement f, peut être observé après f).

Pour éviter ce phénomène, on peut implanter dans l'observateur [JAR 94b] un reséquenceur qui délivrera une trace d'événements préservant la causalité : c'est ce que l'on appelle l'*observation causale*. Il s'agit d'un travail facile puisque, lors de l'observation d'un événement, on connaît le nombre d'événements le précédant sur chaque processus ; il suffit donc éventuellement d'attendre que ses prédécesseurs se soient produits pour traiter l'événement. Le prix à payer est une latence possible dans le diagnostic. En revanche, cela va considérablement simplifier l'algorithme de vérification, qui pourra fonctionner de façon gloutonne (c'est-à-dire qu'il pourra avancer à chaque fois qu'un événement est libéré par le re-séquenceur).

4.2. *Construction du treillis des états*

Obtenir par observation causale une séquence d'événements ne suffit pas en général pour décider de la correction d'une propriété d'ordonnancement. De nombreuses propriétés exigent de considérer l'ensemble des autres entrelacements possibles. C'est notamment vrai lorsque l'on va chercher à vérifier l'ordonnancement causal, ou la concurrence, d'un ensemble d'événements. Ce problème, souvent désigné sous le terme de détection de prédicats globaux dans les systèmes distribués, a été classifié récemment NP-complet [CHA 95].

Il n'est donc pas étonnant que sa résolution ici passe par la construction d'un objet exponentiel en la taille de l'ordre de causalité : le treillis des états. Depuis quelques années, plusieurs algorithmes de construction à la volée du treillis ont vu le jour [COO 91, MED 94, JAR 95], motivés principalement par leur application à l'observation des exécutions réparties. Le principe est le suivant : le treillis est construit incrémentalement, croissant à chaque événement observable traité. La croissance élémentaire pour un pas consiste à dupliquer une partie du treillis déjà construit. Cette partie peut être repérée par un parcours du treillis et par la connaissance des estampilles associées aux événements déjà observés. Le meilleur algorithme connu aujourd'hui a une complexité quasi-linéaire en la taille (nœuds + arcs) du treillis ($\mathcal{O}(\mid I(O) \mid + \mid Arcs(\widetilde{I(O)}) \mid + \mid O \mid^2 n)$ exactement). L'algorithme est d'autant plus coûteux que l'exécution est parallèle (beaucoup d'événements concurrents). C'est une situation bien connue dans le domaine de la vérification des programmes parallèles.

4.3. *Technique de vérification*

Le diagramme de Hasse du treillis des états (voir exemple de la figure 5) $\widetilde{I(O)}$ de l'ordre de causalité $\widetilde{O}$ peut être vu comme un système de transitions : $< I(O), \Sigma, \longrightarrow >$ où l'ensemble des état est $I(O)$, Σ est l'alphabet des actions (les noms des événements observables), et $\longrightarrow$ est défini par les arcs $I \xrightarrow{a} J$ du diagramme.

Pour chaque chemin maximal du treillis, les étiquettes successives rencontrées sur les arcs du graphe forment un mot sur l'alphabet Σ. Ce sont ces mots que nous allons confronter à une propriété globale définie par un automate, afin d'émettre un diagnostic.

Considérons donc, pour illustrer l'approche, que la propriété globale à vérifier est exprimée par un automate fini déterministe. Un tel automate peut être décrit par un 5-uplet $\phi =< \Sigma, Q, q_0, F, \delta >$, où Σ est l'alphabet des actions, Q est l'ensemble des états, q_0 est l'état initial, $F \subseteq Q$ est l'ensemble des états accepteurs, et $\delta : Q \times \Sigma \rightarrow Q$ est la fonction de transition. δ^* est la fermeture transitive de la relation de transition.

Soit $\mathcal{L}(\phi) = \{u \in \Sigma^* \ : \delta^*(q_0, u) \in F\}$ le langage défini par ϕ. Dans notre exemple de la figure 2(b), $\Sigma = \{Req, Ind, Cnf\}$, $Q = \{0, 1, 2\}$, $q_0 = 0$, $F = \{0\}$, et $\mathcal{L}(\phi) = (Req.Ind^+.Cnf)^*$.

La valeur de vérité d'une telle propriété donnée par un automate, est définie par une "relation de satisfaction" $\models$, qui dit à quelle condition une trace d'événements satisfait une propriété ϕ. À toute trace d'événements, on peut associer une état particulier : l'état "terminal" formé par tous les événements observables de la trace. Pour chaque

nœud I du treillis des états, on peut donc considérer la trace (ou préfixe) s'achevant en I, et définir la relation $I \models \phi$.

Soit $Chemins(I)$ l'ensemble des chemins allant du bas du treillis vers le nœud I. En considérant les étiquettes figurant sur les arcs successifs formant les chemins, on peut associer à $Chemins(I)$ un langage fini $\lambda(Chemins(I))$ sur Σ^*, qui définit donc ce que l'on vient d'observer. Pour confronter cette observation au langage généralement de cardinal infini $\mathcal{L}(\phi)$, nous choisissons de dire qu'un nœud I satisfait ϕ si et seulement si un chemin dans $Chemins(I)$ est étiqueté par une séquence acceptée par l'automate ϕ :

$$I \models \phi \text{ ssi } \lambda(Chemins(I)) \cap \mathcal{L}(\phi) \neq \emptyset$$

La question duale, l'inclusion $\lambda(Chemins(I)) \subseteq \mathcal{L}(\phi)$ peut être testée par $I \not\models \neg\phi$, $\neg\phi$ désignant l'automate complémentaire de ϕ.

Des définitions précédentes, on peut aisément déduire que :

$$\forall I \in I(O)\ , I \models \phi \ \Leftrightarrow\ \phi(I) \cap F \neq \emptyset$$

où $\phi(I) = \delta^*(q_0, \lambda(Chemins(I)))$ est l'ensemble des états de l'automate ϕ accessibles par les chemins d'extrémité I.

L'algorithme de vérification va donc essentiellement se ramener à calculer $\phi(I)$ pour l'ensemble des nœuds du treillis. Heureusement, ce calcul va pouvoir être effectué de façon incrémentale : il est facile de voir que $\phi(I)$ peut être calculé au vol pendant la construction du treillis puisqu'il suffit d'attribuer les nœuds du treillis par les ensembles $\phi(I)$, et que le $\phi(I)$ d'un nœud I peut être calculé à partir des attributs des nœuds immédiatement précédents dans le treillis. L'ensemble des nœuds immédiatement précédents existent déjà dans le treillis du moment qu'une observation causale a été reconstruite.

$$\phi(I) = \bigcup_{J : J \xrightarrow{x} I} \ \bigcup_{q \in \phi(J)} \delta(q, x)$$

La figure 7 montre le résultat de la vérification sur notre exemple.

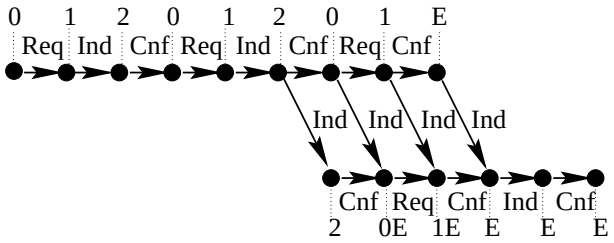

Figure 7. *Les valeurs des $\phi(I)$ sont notées pour chaque nœud I. E est l'état d'erreur de l'automate ϕ. La propriété ϕ est satisfaite dans tous les états étiquetés par un ensemble contenant l'état 0, puisque c'est le seul état accepteur. Elle devient fausse pour toujours à partir du moment où E apparaît dans les étiquettes*

5. Approche de vérification locale

5.1. *Capteurs intelligents*

Jusqu'à présent, nous avons considéré que la vérification s'effectuait "à l'extérieur" du système : une trace d'événements était extraite, puis analysée. Il s'agit maintenant de présenter une méthode d'analyse elle-même répartie, et offrant un service "au vol" : la détection d'une propriété s'effectue à l'instant même où se produit l'événement. Dans ce cas, la détection peut être vue comme un constructeur de "macro-événements". On pourrait encore parler de "capteur intelligent" permettant de structurer (d'abstraire) les événements observables plus microscopiques. Concrètement, nous implantons l'analyse par la superposition d'observateurs locaux sur chaque processus du calcul réparti, et nous allons utiliser les messages du calcul pour accumuler les informations nécessaires à la détection. Le résultat est un algorithme d'estampillage étendu, dans lequel les estampilles véhiculent des informations concernant l'état de la vérification de la propriété à détecter.

La définition de l'algorithme ressemble au développement de la technique globale précédente. La difficulté est due à la distribution de la vérification, qui ne peut être conduite que sur une classe plus simple de propriétés : le séquencement causal de flots d'événements. Nous allons donc considérer une relation de satisfaction différente, qui va porter sur les événements observables eux-mêmes, et non plus sur un état de la trace.

5.2. *Sémantique de flot pour les propriétés*

L'objectif de concevoir une vérification répartie, fonctionnant au vol, impose de rendre la technique incrémentale sur chaque observation d'un nouvel événement ; ce qui impose une localité dans la détection : la vérification va porter uniquement sur l'ensemble des prédécesseurs causaux de x, $\downarrow x$.

Les mots considérés, formés par la succession des étiquettes des événements observables sont les chemins du diagramme de Hasse de l'ordre $\widetilde{O}$. Un chemin d'extrémité x représente le flot des événements qui y conduit. Soit $\mathcal{L}(x)$ cet ensemble. Dans la figure 4(a), on voit que pour le quatrième *Ind*,
$\mathcal{L}(Ind) = \{Req.Ind.Cnf.Req.Ind.Ind.Ind, Req.Ind.Cnf.Req.Ind.Cnf.Req.Cnf.Ind\}$.

Les éléments de $\mathcal{L}(x)$ forment les séquences d'événements qui doivent être effectuées avant x.

Comme dans l'approche globale, on considère que les motifs à détecter sont définis par un automate fini ϕ, et que la satisfaction est déclarée lorsqu'un chemin de l'exécution est reconnu par l'automate.

$$\forall x \in O \ , x \models \phi \Leftrightarrow \mathcal{L}(x) \cap \mathcal{L}(\phi) \neq \emptyset$$

Cette classe de propriétés des exécutions réparties, exprimables par un automate fini, s'avère être assez large, et contient des classes plus simples déjà identifiées, et pour lesquelles il a été publié des algorithmes spécifiques. On peut reproduire notamment l'analogue des "propriétés liées" de [GAR 92], ou des " propriétés atomiques" de [HUR 93], ou encore des "propriétés régulières" de [FRO 94] et [BAB 94].

5.3. *Marquage des événements observables*

Pour un événement observable x, considérons l'ensemble $\phi(x)$ des états atteints par l'automate ϕ lors de l'analyse de tous les mots de $\mathcal{L}(x)$:

$$\phi(x) = \bigcup_{\sigma \in \mathcal{L}(x)} \delta^{\star}(q_0, \sigma)$$

La satisfaction de la propriété ϕ au moment de l'observation de l'événement x se ramène alors à la simple inspection de cet ensemble $\phi(x)$, puisque :

$$x \models \phi \Leftrightarrow \phi(x) \cap F \neq \emptyset$$

En outre, puisque $\mathcal{L}(x)$ peut être calculé incrémentalement, $\phi(x)$ s'exprime récursivement, en fonction des attributs $\phi(y)$ des prédécesseurs immédiats de x :

$$\forall x \in O,\ \phi(x) = \{q_0\} \text{ si } \downarrow^{im} x = \emptyset,\ \phi(x) = \bigcup_{y \in \downarrow^{im} x} \bigcup_{q \in \phi(y)} \delta(q, \lambda(x)) \text{ sinon;}$$

où $\downarrow^{im} x$ désigne les prédécesseurs immédiats de x :
pour $y \in O$, $y \in \downarrow^{im} x$ ssi $y \in \downarrow x$ et $\nexists z \in O \setminus \{x, y\} : y \preceq z \preceq x$.

5.4. *Principe de l'algorithme*

Le calcul de $\phi(x)$ pour chaque événement observable x est distribué sur les processus. L'information distante est acquise par l'intermédiaire des messages. L'automate ϕ est connu de tous les processus. Chaque processus P_i maintient deux tableaux : $LO_i[1..n]$ et $SLO_i[1..n]$. L'idée est qu'à chaque occurrence d'un événement t (observable ou d'interaction), $LO_i[j]$ contient le dernier événement observable sur P_j dans le passé causal de l'événement t. On note $LO_i[j](t)$ et $SLO_i[j](t)$ les valeurs de $LO_i[j]$ et $SLO_i[j]$ au moment de l'occurrence de t. $SLO_i[j](t)$ contiendra l'information d'état $\phi(LO_i[j](t))$, mais seulement si $LO_i[j]$ est maximal parmi l'ensemble des derniers événements observables $\bigcup_{k \in 1..n} LO_i[k]$. $SLO_i[j]$ prendra la valeur $\emptyset$ sinon. Précisément[1] :

$$\begin{array}{rcl}
\forall t \in E_i,\ \forall j \in 1..n & & \\
LO_i[j](t) & = & max_{\preceq}(\downarrow t \cap O_j) \\
SLO_i[j](t) & = & \phi(LO_i[j](t))\ \ si\ \ LO_i[j](t) \in max_{\preceq}(\bigcup_{k \in 1..n} LO_i[k](t)) \\
SLO_i[j](t) & = & \emptyset\ \ sinon
\end{array}$$

Pour un processus P_i, et pour le prochain événement observable x à arriver sur P_i, LO_i présente les n candidats potentiels à être prédécesseurs immédiats de x. Lorsque

[1] Ici, par abus de notation, $\downarrow t$, pour $t \in E$, désigne l'ensemble des prédécesseurs de t, et pas seulement les observables.

x arrive, l'information d'état sera présente dans SLO_i, à condition que l'information associée aux prédécesseurs non immédiats ait été vidée. Le problème de la détection sera alors résolu puisque la bonne valeur de $\phi(x)$ pourra être déduite du contenu de la variable SLO_i :

$$\forall x \in O_i,\ \ \phi(x) = \bigcup_{q \in \bigcup_{j \in 1..n} SLO_i[j](x)} \delta(q, \lambda(x))$$

Le détail de la construction et de la preuve peut être consulté dans [FRO 95]. Le résultat est un algorithme d'estampillage étendu dans lequel les valeurs des SLO_i sont passées dans les messages, en plus des estampilles vectorielles LO_i. Le texte final de l'algorithme est le suivant :

Données:
  $LO_i[1..n]$ *tableau d'entiers;*
  $SLO_i[1..n]$ *tableau d'ensembles d'états;*
Initialisation:
  $\forall j \in 1..n : LO_i[j] := 0;$
  $\forall j \in 1..n : SLO_i[j] := \{q_0\}$;
Quand P_i produit un événement observable x:
  $LO_i[i] := LO_i[i] + 1;$
  $SLO_i[i] := \left\{\delta(q, \lambda(x)), \forall q \in \bigcup_{k \in 1..n} SLO_i[k]\right\}$;
  $\forall j \in 1..n, j \neq i : SLO_i[j] := \emptyset;$ **%** $\phi(x) = SLO_i[i]$ **%**
Quand P_i envoit un message à P_k:
  $LO_i[1..n]$ *et* $SLO_i[1..n]$ *sont passés dans le message;*
Quand P_i reçoit de P_k un message portant LO_k et SLO_k:
  $\forall j \in 1..n$: **faire**
    si $LO_i[j] < LO_k[j]$ **alors** $SLO_i[j] := SLO_k[j]$;
                $LO_i[j] := LO_k[j]$
    sinon si $LO_i[j] = LO_k[j]$ **alors**
      si $SLO_i[j] \neq \emptyset$ *et* $SLO_k[j] = \emptyset$
      alors $SLO_i[j] := \emptyset$

La figure 8 montre la circulation de l'information et le résultat du calcul des étiquettes pour chaque événement observable.

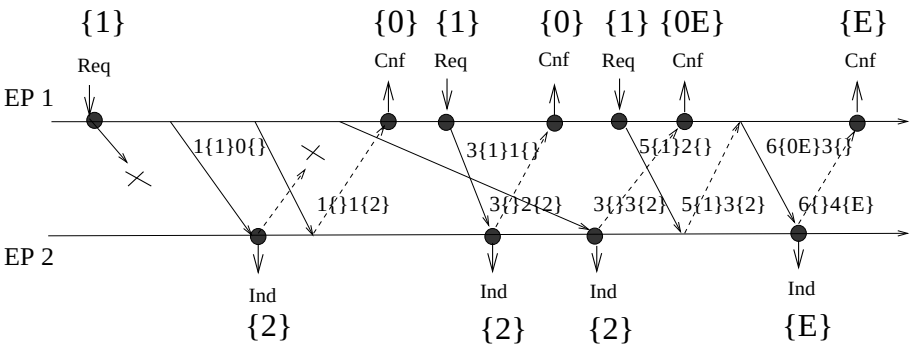

Figure 8. *Analyse au vol : les différentes valeurs des états $LO[1]SLO[1]LO[2]SLO[2]$ passés dans les messages sont figurées. Les valeurs de $\phi(x)$ sont indiquées pour chaque événement observable x*

6. Conclusion

En nous situant dans le contexte du test de l'interopérabilité des processus dans un système réparti, nous avons montré les problèmes et les techniques permettant d'observer les comportements globaux du système et d'effectuer un diagnostic de bon ou mauvais fonctionnement. À cause de la nature répartie des protocoles, ce sont des techniques spécifiques qui sont utilisées, comme la capture au vol des dépendances causales et le calcul des états. Elles s'avèrent nécessaires pour produire un diagnostic fiable dans le cas général. L'exposé présente le meilleur état de l'art du moment, publié essentiellement dans la communauté du débogage réparti. Nous pensons que l'essentiel des concepts est en place, et que le travail prioritaire est d'affiner et spécialiser ces techniques sur de vraies applications. D'ores et déjà, notre équipe s'est engagée dans un processus de valorisation à travers des collaborations industrielles et académiques. Nous avons appliqué ces algorithmes à l'analyse de traces pour évaluer la qualité des exécutions produites par parallélisation automatique de programmes séquentiels (dans le cadre d'un projet inter-PRC et d'un projet européen) et au test d'interopérabilité des réseaux (sur la plateforme de test du Centre Electronique de l'Armement). Ils sont au cœur des outils d'observation implantés dans l'exécutif réparti, appelé POM (Parallel Observable Machine), conçu à l'Irisa. Notre dernière application en cours, utilisant ces techniques, est le diagnostic de pannes dans les réseaux de télécommunications (dans le cadre d'une collaboration avec le CNET).

Remerciements

Ces travaux présentés dans cet article ont été menés dans le cadre des projets PAMPA et ADP de l'IRISA et ont été financés en partie par le Centre Electronique de l'Armement (Celar) et le ministère de la recherche (Projet inter-PRC Traces). Guy-Vincent Jourdan, Eddy Fromentin et Michel Raynal ont participé à l'élaboration des idées et algorithmes qui font l'objet de cette présentation synthétique, initialement préparée pour une conférence invitée des rencontres du parallélisme (RenPar'7, Mons 1995).

7. Bibliographie

[BAB 94] BABAOGLU O. et RAYNAL M., Specification and detection of behavioral patterns in distributed computations , *4th IFIP WG10.4 Int. Conference on Dependable Computing for Critical Application* éd. par Springer Verlag Series San Diego, 1994.

[BAL 92] BALDY P., DICKY H., MEDINA R., MORVAN M. et VILAREM JF., *Efficient reconstruction of the causal relationship in distributed computations*, Rapport technique n ° 92-013, LIRMM, June 1992.

[CHA 95] CHASE C.M. et GARG V.K., Efficient detection of restricted classes of global predicates , *Proc.* 9th *Int. Workshop on Distributed Algorithms, WDAG'95*, Le Mont Saint Michel, France, September 1995.

[COO 91] COOPER R. et MARZULLO K., Consistent detection of global predicates , *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, p. 163–173, Santa Cruz, California, May 1991.

[FID 88] FIDGE J., Timestamps in Message Passing Systems that Preserve the Partial Ordering , *Proc.* 11th *Australian Computer Science Conference*, p. 55–66, February 1988.

[FOW 90] FOWLER J. et ZWAENEPOEL W., Causal distributed breakpoints , 10th *IEEE International Conference on Distributed Computing Systems*, p. 134–141, 1990.

[FRO 94] FROMENTIN E., RAYNAL M., V.K. GARG. et TOMLINSON A.I., On–the–fly testing of regular patterns in distributed computations , *Proc. of the 23rd Int. Conf. on Parallel Processing, St. Charles, IL.*, p. 73–76, 1994.

[FRO 95] FROMENTIN E., JARD C., JOURDAN G.-V. et RAYNAL M., On-the-fly analysis of distributed computations , *Information Processing Letters*, n ° 54, 1995, p. 267–274.

[GAR 92] GARG V.K. et WALDECKER B., Detection of unstable predicates in distributed programs , *Proc. of the 12*th *conf. Foundations of Software Technology and Theoretical Computer Science*, p. 253–264, New Delhi, India, December 1992.

[HUR 93] HURFIN M., PLOUZEAU N. et RAYNAL M., Detecting atomic sequences of predicates in distributed computations , *Proc. of the ACM Conference on Parallel and Distributed Debugging*, p. 32–42, San Diego, May 1993.

[JAR 94a] JARD C., How to observe interoperability at the service level of protocols , *Proc. of the 7th Int. Workshop on Protocol Test Systems, IWPTS'94* éd. par Mizuno T., Higashino T. et Shiratori N., p. 247–262, Tokyo, Japan, November 1994.

[JAR 94b] JARD C., JÉRON T., JOURDAN G.V. et RAMPON J.X., A General Approach to Trace-checking in Distributed Computing Systems , 14th *International Conference on Distributed Computing Systems* éd. par W. Cellary, p. 396–403, Poznan, Poland, June 1994.

[JAR 95] JARD C., JOURDAN G.-V. et RAMPON J.X., On-line computations of the ideal lattice of posets , *Theoretical Informatics and Applications, RAIRO/ITA*, vol. 29, n ° 3, 1995, p. 227–244.

[JAR 96] JARD C. et JOURDAN G.-V., Incremental transitive dependency tracking in distributed computations , *Parallel Processing Letters*, vol. 6, n ° 3, 1996, p. 427–435.

[LAM 78] LAMPORT L., Time, clocks and the ordering of events in a distributed system , *Communications of the ACM*, vol. 21, n ° 7, July 1978, p. 558–565.

[MAT 89] MATTERN F., Virtual time and global states of distributed systems , *Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988* éd. par Cosnard , Quinton , Raynal et Robert , North Holland, 1989.

[MED 94] MEDINA R. et NOURINE L., Algorithme efficace de génération des idéaux d'un ensemble ordonné , *Comptes Rendus à l'académie des Sciences, I*, vol. 319, 1994, p. 1115–1120.

[RAF 91] RAFIQ O., Le test d'interopérabilité des protocoles , *CFIP'91, Ingénierie des protocoles*, Hermes, Septembre 1991.

[SIN 92] SINGHAL M. et KSHEMKALYANI A., An efficient implementation of vector clocks , *Information Processing Letters*, vol. 43, n ° 43, August 1992, p. 47–52.

Titre anglais : Distributed Executions: Detection of Behavioural Properties

Rappel de titre : Observation d'exécutions réparties

Claude JARD est chercheur au CNRS, et effectue sa recherche à l'IRISA depuis 1986. Ingénieur des Télécommunications en 1981, Docteur en 1984, Habilité à Diriger les Recherches en 1994. Il a débuté au CNET de Lannion en 1981, où il a notamment développé l'outil de simulation Véda. Ses travaux portent sur l'ingénierie des protocoles et font la promotion des méthodes formelles pour spécifier, vérifier et tester les logiciels répartis. Il dirige actuellement l'équipe de recherche PAMPA, soutenue par l'INRIA, le CNRS et l'Enseignement Supérieur.