Exécution répartie et temps-réel de réseaux de Petri

Olivier Baldellon, Jean-Charles Fabre, Matthieu Roy

Exécution répartie et temps-réel de réseaux de Petri †

Olivier Baldellon[1,2] and Jean-Charles Fabre[1,2] and Matthieu Roy[1,3]

[1] *CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France ;*
[2] *Univ. de Toulouse, INP, LAAS, F-31400 Toulouse, France*
[3] *Univ. de Toulouse, LAAS, F-31400 Toulouse, France*

L'émergence de systèmes distribués, réactifs à l'environnement et de plus en plus complexes pose de nombreuses problématiques scientifiques. En particulier, malgré les progrès faits pour limiter les erreurs lors de leur conception, la présence de fautes résiduelles est quasiment inévitable du fait de la complexité croissante des systèmes considérés. Il devient nécessaire d'introduire des mécanismes pour gérer les fautes potentielles lors de l'exécution ; on parle alors de supervision. Cet article introduit une approche permettant une supervision distribuée et efficace d'une propriété exprimée sous la forme d'un réseau de Petri.

Keywords: Réseau de Petri, Supervision, Algorithmique répartie

Introduction

Face à la complexité croissante des systèmes informatiques actuels qui peuvent être tout à la fois répartis, réactifs à l'environnement, et modifiés à plusieurs reprises lors de leur existence, les méthodes classiques de développement et de vérification ne suffisent plus à assurer une confiance suffisante dans la fiabilité du système. En effet, une telle complexité rend inévitable la présence de fautes à l'exécution.

L'une des approches possibles pour contenir les fautes résiduelles est de superviser le système s'exécutant pour détecter une éventuelle violation de sa spécification. Concrètement, dans notre approche, la supervision se fait en deux étapes : dans un premier temps, le superviseur capture des évènements système pour vérifier, dans un second temps, que la suite d'évènements détectés correspond à une exécution correcte du système modélisé par un réseau de Petri. Nous nous focalisons dans cet article sur une méthode pour répartir et évaluer efficacement à l'exécution des propriétés exprimées sous forme de réseaux de Petri. Nous décrivons dans une première partie l'idée principale de notre approche avant de la formaliser dans une seconde partie.

1 Exécution spéculative d'un réseau de Petri

1.1 Rappels sur les réseaux de Petri

La supervision compare le comportement mesuré d'un système à un ensemble de comportements autorisés ; il est ainsi nécessaire de modéliser les comportements nominaux du système. Un des formalismes les plus adaptés pour les systèmes répartis temps-réel est celui des réseaux de Petri A-temporisés.

Un réseau de Petri A-temporisé est un graphe représenté par un quintuplet $(P, T, {}^\bullet(\cdot), (\cdot)^\bullet, M_0, I)$ où P représente l'ensemble des places, T l'ensemble des transitions et où ${}^\bullet(\cdot)$ et $(\cdot)^\bullet$ sont des fonctions qui à chaque transition associent un ensemble de places. Un réseau de Petri est un graphe orienté dont l'ensemble des nœuds est $P \cup T$ et dont les arcs sont donnés par les ensembles $\{(p,t) \mid p \in {}^\bullet t\}$ et $\{(t,p) \mid p \in t^\bullet\}$. Chaque place de ce réseau peut contenir des jetons. On appelle marquage une fonction qui à chaque place associe un ensemble de jeton. Le marquage M_0 correspond au

†Avec le soutien de l'ANR, projet MURPHY, contrat ANR-BLAN-SIMI10-LS-100618-6-01

marquage initial. I est une fonction qui à chaque arc entre une place p et une transition t associe un intervalle de temps noté $I(p,t)$. La sémantique d'un réseau de Petri A-temporisé est la même que celle d'un réseau de Petri ordinaire à ceci près qu'un jeton ne peut franchir une transition t à partir d'une place p que si il est resté pendant une durée $d \in I(p,t)$ dans la place p.

1.2 Extension de la sémantique la détection d'erreur

Notre approche, qui sera formalisée dans la partie suivante, consiste à tirer une transition à chaque fois qu'un évènement correspondant est détecté. En d'autres termes, nous n'exécutons pas à proprement parler le réseau de Petri, mais plutôt une observation, ceci afin de vérifier que l'observation correspond bien à une exécution correcte. Concrètement le moniteur reçoit une série d'évènements de la forme $e = (t,\tau)$ où t est la transition qui correspond à l'évènement et τ la date de l'évènement (la notion d'évènement sera complétée et formalisée plus tard) ; au fur et à mesure des réceptions, le moniteur exécute les transitions correspondantes. Pour comprendre plus en détail le mécanisme de notre moniteur, nous allons détailler son comportement sur le réseau de Petri de la Figure 1 à partir de deux observations distinctes.

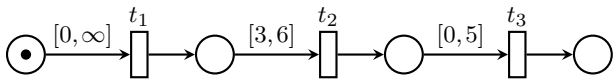

Figure 1: Un réseau de Petri simple

Une premier exemple d'exécution Supposons que le moniteur reçoit dans cette ordre les évènements suivants : $(t_1,10)$; $(t_2,15)$; $(t_3,21)$. Le moniteur va tout d'abord tirer la transition t_1 en notant que le franchissement de cette transition a été fait à l'instant 10. Puis, lorsque le moniteur recevra le second évènement il vérifiera que la transition t_2 est bien franchissable — c'est-à-dire qu'il y a bien un jeton dans chaque place de ${}^\bullet t_2$ — pour ensuite tirer t_2 en comparant les deux dates. Étant donné que $15-10 \in [3,6]$, le moniteur constate pour l'instant la validité de l'exécution. Enfin après la réception du dernier évènement, le moniteur remarque que la transition est bien franchissable mais que $21-15 \notin [0,5]$. Le moniteur tire quand même la transition car l'évènement a bel et bien eu lieu, mais soulèvera une erreur car l'évènement correspondant à t_3 est arrivé trop tard.

Un second exemple d'exécution Supposons cette fois ci que le moniteur reçoit d'abord les évènements $(t_2,15)$ et $(t_3,21)$ dans cet ordre et qu'il n'a pas encore reçu l'évènement $(t_1,10)$. Notre moniteur a toute l'information nécessaire pour détecter l'erreur obtenue lors du franchissement de t_3, sans pour autant être capable de tirer t_3 car il n'y pas de jeton dans la place de ${}^\bullet t_3$: en effet en l'absence de l'évènement $(t_1,10)$ le moniteur ne peut pas tirer t_2.

Une première solution consisterait à attendre l'évènement correspondant à t_1. Cependant, cette solution introduit une latence artificielle à la détection de l'erreur de franchissement de t_3, mais surtout, rien ne nous garantit que l'évènement $(t_1,10)$ sera effectivement reçu. La non détection d'un évènement ne doit pas bloquer l'exécution de notre moniteur.

Plutôt que de rester bloqué dans l'attente de l'évènement correspondant à t_1, le moniteur va tirer la transition t_2 quand bien même cette dernière n'est pas franchissable. Pour cela on modifie légèrement la sémantique du franchissement d'une transition. Au lieu d'enlever les jetons de ${}^\bullet t$ (ce qui n'est possible que si il y a des jetons) le franchissement d'une transition t va correspondre à l'ajout de jetons négatifs dans les places de ${}^\bullet t$ et de jetons positifs dans $t^\bullet$.

L'exemple de la figure 2 montre l'état du réseau de Petri après le franchissement de la transition t_2. Avec un jeton négatif sur la place de ${}^\bullet t_2$ et un jeton positif sur la place de $t_2^\bullet$. Pour vérifier qu'une exécution est correcte il suffit, à chaque fois qu'une place contient un jeton négatif et un jeton positif, de comparer les dates d'apparition de chacun de ces jetons et de vérifier qu'elles sont conformes aux contraintes temporelles.

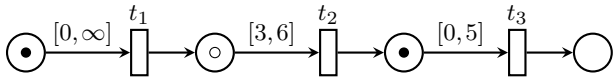

Figure 2: Jetons négatifs

2 Définition formelle et protocole

2.1 Jetons et identifiants

Un jeton est un couple (s, id) où s peut être soit « + » (jeton positif) soit « − » (jeton négatif) et où id représente un identifiant unique du jeton. L'intérêt de la variable id est que lorsque l'on veut ajouter un jeton négatif à un jeton positif, il est nécessaire de savoir quel jeton positif ajouter à quel jeton négatif.

Par exemple, dans l'exemple de la Figure 3, deux jetons positifs sont dans une place. Supposons que le jeton d'identifiant « 1 » est apparu à la date $t = 0$ et que le jeton d'identifiant « 2 » à la date $t = 2$. Supposons alors qu'un jeton négatif arrive à la date $t = 6$. Si ce jeton négatif correspond au franchissement de la transition par le premier jeton, alors il y a une erreur (puisque ce jeton serait resté pendant une durée 6 dans la place). Si par contre il correspond au second jeton, alors il n'y a aucune erreur. Cet exemple montre l'intérêt de distinguer les différents jetons par des identifiants.

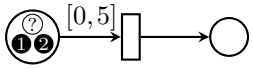

Figure 3: Jetons et identifiants

2.2 Évènements

Nous avons vu dans la partie 1.2 qu'un évènement pouvait être défini par la transition auquel il correspond et par sa date. Nous allons généraliser cette définition pour tenir compte des identifiants.

Dorénavant on définira un évènement par un triplet $e = (t, \tau, M)$ où t est une transition, τ la date de l'évènement en question et M est un marquage. Plus précisément, M est une fonction qui à chaque place de ${}^\bullet t$ associe un jeton négatif (avec un identifiant) et à chaque place de $t^\bullet$ un jeton positif (avec un identifiant).

Soit $\mathcal{E} = e_1 \dots e_n$ une séquence d'évènements avec $e_i = (t_i, \tau_i, M_i)$. On dit que $\mathcal{E}$ est une suite correcte d'évènements si, pour tout $1 \leq i, j, \leq n$ et pour toutes places p et q on a : (1) si $M_i(p) \cap M_j(q) \neq \emptyset$ alors $(i, p) = (j, q)$ et (2) si $|M_i(p)| \cap |M_j(q)| \neq \emptyset$ alors $p = q$. La première condition affirme l'unicité des jetons ; la seconde condition affirme que la version négative d'un jeton apparaît forcément dans la même place que celle de la version positive.

2.3 Le protocole

Notre approche revient à associer à chaque place et à chaque transition un nœud du système. En pratique rien n'interdit plusieurs places ou transitions d'être associées à un même nœud. Chaque place et chaque transition s'exécute dans un *thread* propre sur le nœud associé. Chaque fois qu'une transition reçoit un évènement, elle construit les jetons correspondants et les fait suivre aux places auxquelles elle est connectée (algorithme 1).

Ainsi chaque place, ou plus exactement chaque *thread* associé à une place, va recevoir un ensemble de jeton. Si dans l'ensemble des jetons reçus, il y deux jetons j et $-j$ (de même identifiant mais de signe opposé) alors il suffit de comparer leurs dates pour savoir si une erreur s'est produite. Au cas où la version négative d'un jeton ne serait pas reçue, lors de la réception d'un jeton dont l'identifiant est nouveau, une fonction TIMEOUT est lancée dans le but de déclencher une erreur si la version négative du jeton n'est pas reçue au bout d'un temps suffisamment long pour être sûr qu'une erreur s'est produite (que le jeton soit manquant ou arrivé trop tard). Cette valeur peut

Algorithme 1 Protocole des places et des transitions

```
1: TRANSITION(Pre, Post)
2: Lors de la réception de e = (t, τ, M)
3: pour chaque place p de Pre ∪ Post faire
4:     p ! (t, τ, M(p))
5: fin pour
```

```
1: PROCÉDURE(I)
2: Lors de la réception de (t, τ, token)
3: si il existe (t', τ', −token) ∈ MÉMOIRE alors
4:     MÉMOIRE ← MÉMOIRE − (t', τ', −token)
5:     si token > 0 alors ok = (τ' − τ) ∈ I(p, t')
6:     sinon ok = (τ − τ') ∈ I(p, t)
7:     fin si
8:     si ok = FAUX alors erreur détectée
9: sinon
10:    MÉMOIRE ← MÉMOIRE + (t, τ, token)
11:    TIMEOUT(token,t)
12: fin si
```

être déterminée facilement dans un système synchrone où les délais entre l'émission et la réception d'un message sont bornés.

3 État de l'art

Certains travaux ont déjà été entrepris dans l'optique d'obtenir un mécanisme de supervision distribué ; citons par exemple [ZSLL09] et [JRR94]. Cependant, la mise en place de la distribution du calcul est faite différemment. Dans notre approche, les propriétés à vérifier sont représentées par un unique réseau de Petri que nous exécutons ensuite de manière répartie. Au contraire, ces travaux attribuent un moniteur à chaque propriété. Dans le cas de [JRR94], cette approche se révèle être particulièrement efficace car ils se restreignent aux propriétés exprimables par des conjonctions de propriétés de bases. Si cette approche permet de répartir le calcul au mieux, elle souffre d'un manque d'expressivité (incapacité de faire un simple « ou logique ») là où les réseaux de Petri A-temporisés se révèlent au contraire très riches.

D'autres travaux remarquables sur la supervision asynchrone [FBHJ05] ou temps-réel [CJ05] sont ceux de C. Jard *et al.*. Leur approche est cependant profondément différente de la notre dans le sens où chaque évènement n'est pas associé à une mais à plusieurs transitions ; leurs travaux consistent alors à trouver les exécutions compatibles avec l'observation du système.

Il existe une autre approche pour calculer de manière distribuée l'exécution d'un réseau de Petri présentée dans [Hop91]. À la différence de nos travaux, l'exécution est faite de manière séquentielle ; une transition ne peut être franchie que si elle est déjà franchissable et donc il est nécessaire d'attendre le franchissement des transitions qui la précèdent logiquement.

Conclusion

Ce papier introduit une nouvelle méthode d'exécution de réseau de Petri dont la principale originalité est l'introduction des jetons négatifs. Cette approche permet d'éviter que la non détection ou la détection tardive d'un évènement par le système ne bloque ou ralentisse l'éxècution du moniteur. De plus notre protocole est naturellement distribué et ne nécessite aucun nœud central.

Références

[CJ05] T. Chatain and C. Jard. Time supervision of concurrent systems using symbolic unfoldings of time petri nets. *Formal Modeling and Analysis of Timed Systems*, pages 196–210, 2005.

[FBHJ05] E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1) :33–84, 2005.

[Hop91] R. Hopkins. Distributable nets. *Advances in Petri Nets 1991*, pages 161–187, 1991.

[JRR94] F. Jahanian, R. Rajkumar, and S.C.V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3) :247–273, 1994.

[ZSLL09] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. DMaC : Distributed Monitoring and Checking. *Lecture Notes in Computer Science*, 5779 :184, 2009.