Supervision en ligne de propriétés temporelles dans les systèmes distribués temps-réel

Olivier Baldellon

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Systèmes Informatiques

Présentée et soutenue par :

M. OLIVIER BALDELLON

le vendredi 7 novembre 2014

Titre :

SUPERVISION EN LIGNE DE PROPRIETES TEMPORELLES DANS LES SYSTEMES DISTRIBUES TEMPS-REEL

Ecole doctorale :

Systèmes

Unité de recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (L.A.A.S.)

Directeur(s) de Thèse :

M. JEAN CHARLES FABRE

M. MATTHIEU ROY

Rapporteurs :

M. KAMEL BARKAOUI, CNAM PARIS

M. MICHEL RAYNAL, UNIVERSITE RENNES 1

Membre(s) du jury :

M. MICHEL RAYNAL, UNIVERSITE RENNES 1, Président

M. CLAIRE PAGETTI, ONERA TOULOUSE, Membre

M. JEAN CHARLES FABRE, INP TOULOUSE, Membre

M. MATTHIEU ROY, LAAS TOULOUSE, Membre

M. THOMAS ROBERT, TELECOM PARISTECH, Membre

Supervision en ligne de propriétés temporelles dans les systèmes distribués temps réel

L'œil était dans la tombe et regardait Caïn

La Légende des Siècles — La Conscience
Victor Hugo

Sommaire

Chapitre I

Introduction

> Le serpent dit à la femme : « Non, vous ne mourrez pas, mais Dieu sait que le jour où vous en mangerez, vos yeux s'ouvriront et vous serez comme des Dieux possédant la connaissance de ce qui est bon ou mauvais »
>
> *La Genèse*
> La Bible

En présence de systèmes critiques de plus en plus complexes, la problématique de la sûreté de fonctionnement prend chaque jour toujours plus d'importance. En effet, devant l'avènement d'une informatique distribuée, temps réel et en constante interaction avec son environnement, il devient toujours plus difficile de garantir une confiance suffisante dans les systèmes considérés. La problématique de la sûreté de fonctionnement ne peut évidemment être résolue que par la convergence d'approches aussi nombreuses que diverses :

- la réduction des erreurs de conception ;
- la gestion transparente des erreurs via différents mécanismes de tolérance aux fautes ;
- la détection d'erreurs à l'exécution via un superviseur ;
- la mise en place de politique de recouvrement en cas d'erreurs détectées.

Dans cette thèse, nous nous concentrerons tout particulièrement sur la mise en place d'un superviseur. On peut considérer deux catégories de superviseurs : les superviseurs bas niveau (que nous appellerons capteurs) dont l'objectif est de faire des mesures sur le système et les superviseurs haut niveau dont le but est de vérifier, à partir de mesures, que le système satisfasse une spécification donnée. C'est cette deuxième catégorie de superviseur à laquelle nous nous intéresserons.

Avant de rentrer dans les détails il est nécessaire de commencer par décrire le contexte et l'objectif de nos travaux.

1.1. Différentes approches pour la sûreté de fonctionnement

Comme nous avons eu l'occasion de le dire dans l'introduction, la sûreté de fonctionnement est un champ scientifique vaste qui combine de nombreuses approches. Elle est définie dans le *Guide de la sûreté de fonctionnement* de Jean-Claude Laprie *et al* [Lap+95] par :

> « La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. »

Ces approches peuvent se répartir en trois grands domaines. Dans la suite, les fautes peuvent être tout autant matérielles, logicielles qu'humaines.

Élimination des fautes C'est l'approche première et la plus évidente. Si l'on veut un système sûr de fonctionnement, il faut en éliminer les fautes. Cette élimination peut être faite durant la conception ou après la conception via de nombreux tests. Malheureusement dans des systèmes distribués, temps réel et en constante interaction avec leur environnement, il est illusoire et impossible de s'assurer de l'inexistence de fautes. Les modélisations de l'environnement et du matériel ne sont jamais exactes car elles correspondent par définition à des simplifications. Enfin, il existe de nombreux théorèmes informatiques qui montrent l'impossibilité de vérifier des propriétés non triviales sur des logiciels ; en particulier nous pouvons citer le fameux problème de l'arrêt d'Alan Turing. Si cette étape est donc fondamentale, elle n'est pourtant pas suffisante.

Tolérance aux fautes La tolérance aux fautes peut être faite de deux façons, souvent complémentaires. La première approche consiste à mettre en place des systèmes qui supportent de manière transparente les différentes fautes rencontrées, et ceci quelque soit leur nature. Cet objectif peut être atteint via la redondance associée à divers algorithmes (Paxos [Lam01] pour prendre un exemple) ; si sur dix ordinateurs, un tombe en panne, le système continuera à fonctionner sans que le client ne voie une quelconque différence de qualité dans le service rendu.

Cependant, toutes les fautes ne peuvent pas être traitées de manière transparente ; il est alors nécessaire d'agir en informant un opérateur humain ou en dégradant le service.

Dans ces deux approches (transparence ou dégradation) un superviseur peut être utile. En effet, le rôle de cet outil est de détecter les erreurs pour permettre de décider un recouvrement, qu'il soit transparent ou dégradé. Cet outil n'est pas suffisant mais se doit d'être utilisé de manière complémentaire à d'autres services, de traitement de fautes en particulier.

Prévision des fautes La prévision des fautes est une approche d'analyse du système qui vise à estimer la présence, la création et les conséquences des fautes ou erreurs sur la sûreté de fonctionnement, et qui peut permettre, par contrecoup, de chercher à les éliminer ou à les tolérer.

1.2. Superviseur

Nous définissons ici précisément ce que nous entendons par superviseur. Le terme est très général et possède ainsi de nombreux sens distincts allant du simple observateur à un véritable cerveau contrôlant un système entier. En effet, il ne suffit pas d'observer, il faut interpréter ces observations distribuées géographiquement ; il faut être capable de vérifier autant que possible un nombre maximal de propriétés à partir d'un minimum d'information. C'est cette définition que nous retiendrons dans cette thèse. Nous allons donc nous concentrer dans ce manuscrit sur les superviseurs distribués et temps réel. Par la suite, nous considérerons comme synonyme les expressions « moniteur » et « superviseur ».

Afin de clarifier ce que nous entendons par là et ce que nous voulons faire, nous détaillerons les trois mots de l'expression « moniteur temps réel distribué ». Un lecteur intuitif aura compris que derrière ces trois notions s'en cache une quatrième tout aussi importante : la robustesse.

Moniteur (ou superviseur) Un moniteur est avant tout un outil dont le but est d'observer un système et d'en extraire les informations désirées. Il n'a souvent qu'une vision partielle des évènements et une vision superficielle du système, système qui n'est souvent, aux yeux du moniteur, composé que de « boîtes noires ». Il doit de plus considérer les interactions entre ces différents composants. Mais, cependant, il ne suffit pas d'observer ; il faut comprendre, interpréter et faire des choix. Nous nous concentrerons dans ce manuscrit sur la détection d'erreurs, c'est-à-dire les comportements non désirés du système observé, par notre moniteur.

Temps réel Si ces informations ont pour raison d'être la mise en place d'un système de rétroaction, il est nécessaire que le moniteur soit temps réel. La rétroaction consiste simplement à utiliser le diagnostic fourni par notre moniteur pour agir sur le système supervisé. Par exemple, si le système s'exécute de manière plus lente que la normale suite à une charge trop élevée, il peut être intéressant de dégrader le service plutôt que de dépasser un délai. Cependant pour que le mécanisme de rétroaction soit efficace, il est nécessaire de poser le diagnostic au plus tôt ; il est donc nécessaire dans ce cas d'avoir un moniteur vérifiant certaines contraintes temporelles.

Distribué Une façon simple de mettre en place un moniteur consiste à centraliser les informations sur un serveur central qui par de lourds calculs pourra décider d'un diagnostic. Cependant une telle approche est à proscrire pour au moins trois raisons, chacune suffisante seule.

1) *Un point unique de défaillance.* La centralisation est fragile. Il suffit que le serveur tombe en panne pour que le moniteur ne marche plus. Une approche plus décentralisée permet une plus grande robustesse.
2) *La latence.* La centralisation induit des délais dans le diagnostic. En effet, supposons qu'un problème ait lieu dans un recoin sinistré et lointain de notre système. Il faut que l'information transite jusqu'au moniteur au risque de perdre de l'information en route, risque d'autant plus grand que le chemin à parcourir est long. Si, de plus, le moniteur décide de lancer une procédure de recouvrement, il faut que les consignes retraversent tout le sys-

tème pour en rejoindre l'extrémité. Le diagnostic sera potentiellement mauvais, la boucle de rétroaction sera lente et les consignes risquent alors de ne pas arriver à temps.

3) *L'inadéquation au système.* La centralisation une solution trop simpliste pour être théoriquement élégante. Si le lecteur n'est pas convaincu par les arguments de type esthétique nous pourrons ajouter que cette simplicité, résultant en une sous utilisation des ressources du système, a des répercutions sur le coût de la supervision. En effet, dans un système réparti, nous avons une puissance de calcul distribuée ; autant en profiter pour effectuer le travail de supervision lorsque cela est possible, plutôt que d'ajouter un serveur central. D'autant plus qu'un serveur central est clairement un goulot d'étranglement qui passe mal à l'échelle.

Pour toutes ces raisons, il est nécessaire de répartir notre superviseur. Cependant, cette distribution apporte de nombreuses questions. Il faut être capable de vérifier les propriétés le plus localement possible afin de gagner en rapidité et en efficacité. Il faut donc réussir à router les communications pour réussir à agréger l'information nécessaire en un point donné du réseau. Une des conséquences de cette décentralisation est que l'état du système, du moins tel que notre moniteur se le représente, n'est plus forcément accessible globalement ; la vision du système doit malgré tout former un tout cohérent même si elle n'est pas centralisée.

Robustesse Le moniteur étant une pièce maîtresse dans la politique de tolérance aux fautes, il est fondamental qu'il soit un des composants les plus robustes du système. Il est nécessaire qu'il puisse se décider même en cas d'information manquante et qu'il soit capable de détecter la disparition d'une partie du système observé. Cette dernière condition peut sembler évidente, mais puisque, comme nous venons de l'expliquer, nous voulons mettre, par la distribution, le moniteur au plus proche du système, il se peut que lors de la disparition d'un sous-système, la partie de moniteur en charge de ce sous système tombe en panne. Ceci est évidemment à éviter à tout prix.

1.3. Modèles, évènements et corrections

L'objectif de cette thèse est donc de mettre en place un superviseur robuste capable de vérifier des propriétés complexes en temps réel et de manière distribuée. Un dernier point sur lequel nous passerons davantage de temps dans le prochain chapitre concerne l'observation. « Observation », comme « supervision » est un mot suffisamment général pour prendre un grand nombre de sens. Précisons alors ce que nous entendons par « observation ».

Notre moniteur aura une représentation interne du système, représentation qu'il fera évoluer à chaque observation. Les observations seront toutes des évènements de la formes $e = (t, \tau)$ où t est la transition correspondant à l'évènement (par transition, on entend un nom ou une référence indiquant comment l'état doit évoluer) et τ, la date de l'évènement.

La robustesse de notre moniteur se traduit par la capacité à fournir des diagnostics pertinents même si certains évènements ne sont pas observés. À partir de maintenant on s'impose une règle fondamentale dans le cas d'un moniteur :

Définition 1 (Correction d'un moniteur).

Soit un système supervisé par un moniteur. Ce moniteur est dit correct s'il vérifie les deux propriétés suivantes :

1) quelle que soit la qualité de l'observation, si le système ne vérifie pas ses spécifications une erreur doit être levée.
2) si l'exécution est correcte, si l'observation est parfaite et si les hypothèses de synchronie sur les délais d'acheminement des messages sont respectées alors aucune alarme ne doit être déclenchée.

À contrario, on s'autorise si nécessaire à lever une erreur même si le système se comporte de manière correcte (en particulier, en cas de problèmes d'observation ou de réseau momentanément asynchrone)

Pour faire simple un moniteur peut faire deux types d'erreurs : soit il déclenche un fausse alarme (ce qui en soit n'est pas très grave) soit il ne détecte pas une erreur (ce qui est beaucoup plus grave). Un moniteur est dit correct s'il déclenche au moins une alarme pour chaque erreur. Cependant cette propriété de complétude seule est trop simple. En effet, un moniteur qui n'aurait de cesse de déclencher des alarmes sans se soucier de l'observation ne laisserait passer aucune erreur et résoudrait donc le problème ! Il nous faut donc considérer que les alarmes lancées par un moniteur ne doivent se lancer que si quelque chose ne s'est pas passé correctement (propriété de précision), comme une erreur dans l'exécution du système, dans son observation ou dans les hypothèses concernant les délais d'acheminement de messages (si ces hypothèses ne sont pas vérifiées, on dit que le réseau est asynchrone).

1.4. Contenu du manuscrit

Ce document de thèse, à l'exception de l'introduction et de la conclusion, est constitué de deux grandes parties. Les chapitres II, III et IV présentent les bases théoriques de notre moniteur. Les deux chapitres suivants, V et VI, détaillent la mise en pratique de notre approche.

Théorie Pour être capable de vérifier que le fonctionnement du système est conforme à ses spécifications, notre moniteur se doit de se représenter les propriétés à vérifier ainsi que l'état du système à travers un formalisme donné. Dans cette thèse, nous utiliserons deux formalismes ; le second étant une généralisation du premier. Le chapitre II présente ainsi brièvement les deux différents modèles utilisés dans cette thèse ainsi que leurs avantages et intérêts et présente aussi les travaux connexes existants. Chacun de ces modèles et leurs algorithmes de supervision associé sera détaillés dans les chapitres III et IV.

Implémentation La mise en place pratique d'un superviseur ne se limite pas à un algorithme exécutant un modèle. Cette thèse consacrera donc le chapitre V sur la modélisation concrète de

certaines propriétés générales, sous la forme d'une bibliothèque de propriétés simples et combinables entre elles. L'objectif est de montrer, via un certain nombre d'exemples représentatifs, les fondements d'une telle bibliothèque. Enfin le chapitre VI détaillera notre implémentation.

Chapitre II

Contexte, état de l'art, modèle et supervision

> Le seul véritable voyage, le seul bain de Jouvence, ce ne serait pas d'aller vers de nouveaux paysages, mais d'avoir d'autres yeux, de voir l'univers avec les yeux d'un autre, de cent autres, de voir les cent univers que chacun d'eux voit, que chacun d'eux est ; et cela, nous le pouvons avec un Elstir, avec un Vinteuil ; avec leurs pareils, nous volons vraiment d'étoiles en étoiles.
>
> *À la recherche du temps perdu — La Prisonnière*
> Marcel Proust

Après avoir détaillé ce que nous attendons d'un moniteur, il nous faut encore préciser l'architecture globale que l'on se donne pour faire tourner le superviseur. Il nous faut aussi discuter des modèles utilisés pour exprimer les propriétés que l'on souhaite vérifier et enfin nous discuterons nos choix en les comparant avec l'état de l'art.

En pratique, nous n'allons pas détailler un seul modèle mais deux, le second étant une généralisation du premier. Comprendre la distinction entre ces modèles est important dans le sens où ils feront l'objet des chapitres III et IV.

2.1. Supervision : des nœuds et des capteurs

Dans un premier temps, détaillons l'infrastructure sur laquelle nous allons faire tourner notre moniteur. Comme nous avons eu l'occasion de le dire dans l'introduction, un superviseur n'observe pas directement le système ; l'observation directe pouvant être faite de manière locale et ponctuelle, nous considérons qu'elle ne pose pas de problème. Ce que nous appelons système est en fait constitué d'un ensemble de capteurs capables de générer des événements. En fait, notre superviseur ne voit le système physique, réel, qu'à travers les différents capteurs. Pour

être exact, chaque capteur n'est responsable que d'un évènement. Si le capteur détecte l'évènement (une porte qui s'ouvre, une température descendant en dessous d'une valeur critique) il envoie au moniteur un message de la forme (t, τ) où t est l'identifiant de l'évènement et τ la date de sa dernière occurrence.

Le superviseur est lui-même distribué ; il est constitué d'un ensemble de nœuds. Le système et le superviseur communiquent à sens unique, des capteurs vers les nœuds, comme le montre la figure II.1. Chaque nœud peut ensuite déclencher si nécessaire une alarme. La communication pourrait se faire dans les deux sens si l'on désire implémenter une boucle de rétroaction ; ce sujet n'est cependant pas considéré dans ce manuscrit.

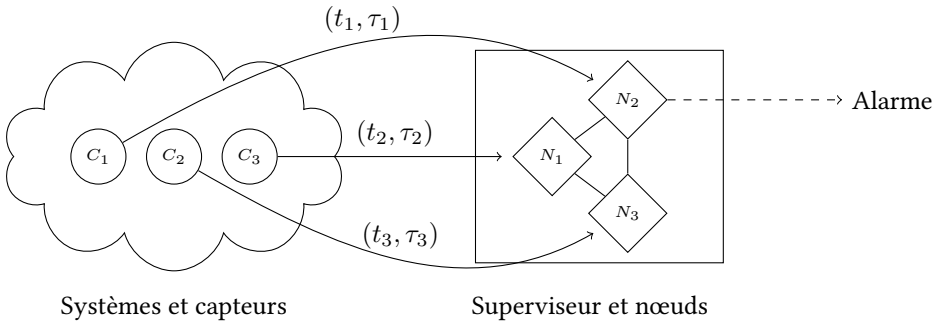


Figure II.1.: La supervision : notre approche

2.2. Un exemple de système : le déplacement d'un robot

Plus qu'un système physique, un moniteur est avant tout un logiciel distribué dont l'objectif est de vérifier certaines propriétés. Pour cela il se doit d'avoir une représentation de l'état du système. Nous allons dans cette partie considérer un robot se déplaçant dans un bâtiment constitué de quatre salles, comme montré dans la figure II.2. Différents capteurs sont localisés dans les portes qui permettent de passer d'une pièce à l'autre. Par exemple, on pourrait chercher simplement à vérifier la propriété : « Le robot doit franchir la porte qui mène de la pièce A à la pièce B au moins toutes les cinq minutes »

Il est important de prendre conscience que notre but n'est pas de diriger le robot afin de faire en sorte que la propriété reste vraie — bien que fondamentalement cela serait faisable avec des mécanismes de rétroaction — mais simplement de vérifier que le comportement du robot satisfasse bien à ses spécifications.

A	B
D	C

Figure II.2.: Le bâtiment

2.3. Modélisation du système avec les transitions évènementielles

Le système présenté ci-dessus constitué du robot se déplaçant dans le bâtiment peut-être représenté par un réseau de Petri. Si le lecteur n'est pas familier avec un tel formalisme, nous l'invitons à se référer à la section 3.1. Pour faire simple, un réseau de Petri est simplement un graphe dans lequel se déplace un ou plusieurs jetons. Les jetons sont sur les places — représentée par des cercles — et doivent franchir des transitions — représentées par des rectangles. Le franchissement d'une transition se fait en supprimant les jetons placés en amont de cette transition et en créant des jetons dans les places situées en aval. Appliqué à notre exemple, le réseau de Petri correspondant est représenté figure II.3.

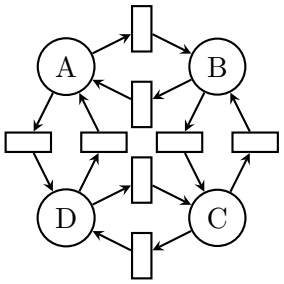


Figure II.3.: Le réseau de Petri associé

Dans cet exemple, chaque transition correspond à un capteur. Le superviseur n'a qu'à tirer les transitions correspondantes à chaque fois qu'un évènement est reçu. C'est pourquoi l'on parle de transition évènementielle.

2.4. Modélisation de propriétés avec les transitions logiques

Cependant, en se limitant à cette sémantique où l'on tire une transition seulement lorsqu'un évènement envoyé par un capteur est reçu, on obtient un modèle facilement exécutable mais malheureusement faiblement expressif. Par exemple, considérons deux robots se déplaçant dans le bâtiment ; l'état du système peut facilement être représenté par deux réseaux de Petri correspondant à ceux de la figure II.3, un pour chaque robot.

Supposons maintenant que l'on désire vérifier que le comportement des robots est bien synchronisé via la propriété suivante :« Le passage du premier robot par la transition "CB" doit être synchronisé à plus ou moins cinq unités de temps avec le passage de la transition "AD" du second robot ». Cette propriété peut être représentée par le réseau de Petri de la figure II.4, dans lequel nous associons à chaque robot un sous-réseau de Petri contenant les places A, B, C et D.

Le principe est relativement simple : lorsque la transition "CB" du premier réseau de Petri est franchi, un jeton est ajouté dans la place p ; ce jeton doit franchir la transition colorée dans l'intervalle de temps $[0, 5]$ après son apparition dans cette place. Cependant un telle transition ne pourra être franchie que si un autre jeton se trouve dans la place q. En conséquence, il

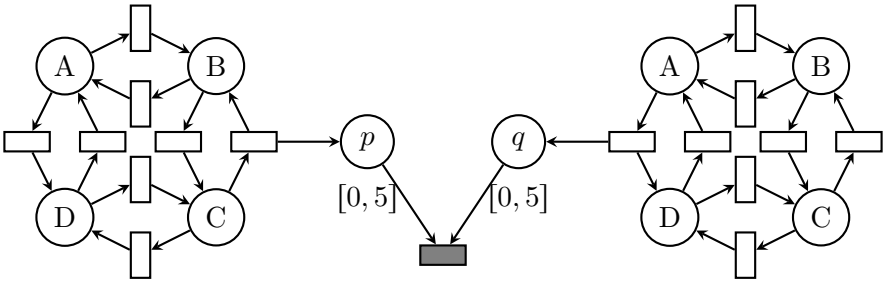


Figure II.4.: Le réseau de Petri associé

faut que dans les cinq unités temps suivant le franchissement de la transition "CB" un jeton apparaisse dans la place q. Cela n'est possible que si "AD" a été franchie dans le second réseau de Petri moins de cinq secondes avant le franchissement de "CB" ou moins de cinq secondes après. Dans tous les cas, la transition finale n'est franchissable que lorsque les deux transitions sont franchies à plus ou moins cinq secondes d'écart.

Cependant, on peut remarquer qu'aucun évènement ni aucun capteur ne correspond à cette transition. C'est pour cette raison qu'elle est représentée en gris. Elle correspond à ce que l'on appelle des transitions logiques. Ces transitions ne sont pas franchies en fonction des observations des capteurs, mais dès lors que le franchissement est possible. Dans ce cas précis, elle est franchissable à partir de la date τ (si elle existe) vérifiant :

1) un jeton est apparu dans la place p a une date que nous noterons τ_p
2) un jeton est apparu dans la place q a une date que nous noterons τ_q
3) $\tau - \tau_p \in [0, 5]$
4) $\tau - \tau_q \in [0, 5]$

Une définition plus précise et plus générale de la sémantique associée à ces transitions sera introduite section 4.1. Si les transitions évènementielles sont utiles pour représenter l'état du système et vérifier certaines propriétés simples concernant cet état, les transitions logiques permettent de modéliser des propriétés plus complexes.

2.5. Modèles d'observations

Nous avons vu dans la partie 1.3 que l'objectif d'un superviseur était d'exécuter un modèle à partir d'évènements. Nous avons introduit dans les parties 2.3 et 2.4 différents modèles — tous basés sur les réseaux de Petri temporisés — pour exprimer des propriétés plus où moins expressives.

Nous allons dans cette partie introduire brièvement les hypothèses que l'on peut faire sur l'observation d'évènements. En effet un superviseur se voulant être une pièce maîtresse de la sûreté de fonctionnement d'un système, il doit être capable de fonctionner même en cas d'observations imparfaites. Nous considérerons donc trois types d'erreurs.

Les retards Bien évidemment, nous devrons considérer l'éventualité de messages reçus avec retard. Cependant, si nous voulons faire du temps réel, nous ne pouvons pas faire l'hypothèse d'un système asynchrone. Nous sommes donc dans l'obligation de considérer un retard maximal possible au-delà duquel une erreur devra être soulevée. Tant que le système sera synchrone, le diagnostic de notre moniteur devra être parfait. Cependant, en accord avec la définition 1, si le système devient asynchrone, on s'autorisera à déclencher une erreur en cas de messages reçus trop tardivement.

Omission Nous devons aussi considérer l'éventualité de perdre des messages. Avec une observation incomplète, il est difficile, si ce n'est impossible de vérifier correctement toutes les propriétés spécifiées. Une message omis peut l'être pour deux raisons : 1) soit l'évènement n'a jamais eu lieu, 2) soit le message associé s'est perdu. Puisque ces deux scénarios sont indistinguables, on parlera dans les deux cas d'erreur d'omission. Cela ne pose pas de problème, car la définition 1 autorise à notre moniteur à lancer de fausses alarmes (s'il n'a pas assez d'information pour être certain que la propriété est vérifiée) mais nous interdirons les faux négatifs, c'est-à-dire les erreurs non détectées.

Désordre Enfin, les retards détaillés plus haut peuvent avoir pour conséquence la réception dans le désordre d'évènements. Il est important de pouvoir détecter une erreur au plus tôt même avec un aperçu partiel de la liste d'évènements. Il ne faut donc pas, pour tenir compte d'un évènement, faire l'hypothèse que tous les messages précédents sont déjà reçus. Cette problématique sera détaillée sur un exemple de la partie 3.2.

2.6. Travaux similaires

La supervision, éventuellement distribuée et temps réel, est un domaine de recherche particulièrement actif. Le lecteur intéressé pourra se référer à deux rapports faisant l'état de l'art de la discipline, l'un jusqu'en 1992 dans le cas distribué [MSS93], l'autre jusqu'en 2009 dans le cas distribué et temps réel [GP10].

Dans un premier temps nous parlerons d'outils de supervision spécifiques à un problème donné. Nous présenterons ensuite plusieurs travaux dans le domaine des superviseurs centralisés. Puis, nous nous concentrerons alors sur deux grands formalismes utilisés dans ce domaine, les logiques temporelles et les réseaux de Petri. Nous terminerons enfin par plusieurs exemples montrant les utilisations potentielles de moniteurs.

2.6.1. Supervision

Supervision centralisé

De nombreux travaux ont été faits concernant la supervision. Parmi les plus outils les plus notables, le lecteur pourra se référer à MAC : *Monitoring and Checking* [KVBAKLS99], *Javapath explorer* [HR01 ; HR04], MOP : *Monitoring Oriented Programmation* [CR07] ainsi que RV [MR10] basé sur le précédent. Si tous ces outils sont écrits en Java pour superviser des

programmes Java, il existe aussi des outils de supervision basés sur le langage C, comme par exemple RMOR [Hav08].

Tous ces outils ont un mode de fonctionnement relativement similaire. Le code que l'on désire monitorer est préalablement annoté, ces annotations servant ensuite à générer un moniteur.

Supervision décentralisée

Pour vérifier la conformité de certains protocoles distribués, comme TCP, Bhargavan et collab. [BCMG01 ; BG02] utilisent un moniteur qui, observant les échanges de messages entre le client et le serveur, mime l'exécution du protocole en faisant évoluer un automate. Si une telle approche permet de surveiller l'implémentation d'un protocole, elle n'est pas forcément adaptée pour des propriétés plus générales qui demanderont l'accès à des évènements internes au système qu'il faudra explicitement envoyer au moniteur.

Temps réel et observation

Le cas des logiques temporelles et des réseaux de Petri étant traité plus tard, nous citerons ici les travaux de Robert et collab. [RFR08] basés sur le formalisme des automates temporisés dont l'objectif est de détecter les erreurs au plus tôt. Les automates temporisés n'étant que difficilement distribuables, cette approche ne peut être utilisée telle quelle dans un contexte réparti.

Le problème de la majorité des approches que nous avons vues et que nous allons voir est qu'elles sont souvent invasives. Une méthode utilisable pour détecter des évènements sans coût ni intrusion dans le système observé consiste à brancher un FPGA directement sur le bus PCI. Ainsi, le moniteur observe sans interférence. Cette approche, par sa discrétion, est particulièrement intéressant dans le domaine du temps réel dur. Elle est utilisée autant dans les travaux de Pellizzoni et collab [PMCR08] que dans les travaux de Kritikakou et collab [KBPRR14 ; KBPRRV13]. Notons que si nous voulons un système de supervision distribué n'interférant pas avec le système, il faut aussi utiliser des canaux de communication différents pour le moniteur et pour le système.

2.6.2. Logique temporelle

Parmi les travaux sur les moniteurs de propriété temps réel, beaucoup sont basés sur de la supervision centralisée de propriétés exprimées par des formules de logique linaire temporelle, comme par exemple *Linear Temporal Logic* : LTL.

Parmi ces travaux, on peut citer [MR10 ; BLS06]. De même pour s'adapter à la supervision dans laquelle la taille des traces est bornée, différentes variantes de LTL, comme par exemple past-time LTL [LMS02 ; HR02], ont été étudiés. Dans le cas des logiques linaires, l'approche reste fondamentalement centralisée ce qui rend l'adaptation pour des moniteurs distribués difficiles. En effet, généralement, à chaque formule LTL est associé pour une évaluation à l'exécution un automate temporisé équivalent. Comme un automate temporisé a un état qui ne possède pas de décomposition naturelle en sous états, ces approches ne peuvent pas être facilement distribuées, bien que cela commence à être étudié comme nous allons le montrer par la

suite..

Dans ces formalismes, on utilise en général trois valeurs pour le diagnostic : vrai, faux et ne se décide pas. La dernière de ces valeurs correspond au cas où pour l'instant aucune erreur n'a été détectée mais dont une future observation pourra rendre la propriété vérifiée incorrecte.

Dans notre thèse, nous avons pourtant choisi de n'utiliser que deux valeurs : vrai si aucune erreur n'a été détectée (symbolisé par l'absence d'alarme) ; faux si une erreur a été détectée ou s'il n'est pas possible de garantir l'absence d'erreur, suite par exemple à une observation imparfaite.

Logique temporelle temps réel

Malgré leur nom, les logiques temporelles ne permettent pas d'exprimer des propriétés temps réel ; en fait, elles permettent d'exprimer des liens de causalité asynchrones. Par exemple, avec φ et ψ des formules logiques, on peut exprimer « À partir de maintenant φ sera toujours vrai », ou encore « φ sera vrai tant que ψ ne l'est pas ».

Il faut généraliser le formalisme pour tenir compte de délais temps réel. Parmi ces formalismes, nous pouvons citer TLTL (*Timed* LTL) [ABLS05 ; BLS06]. TLTL est simplement la logique LTL à laquelle on ajoute deux opérateurs qui permettent d'exprimer :

- que le délai écoulé depuis la date de la dernière occurrence d'un évènement est dans un intervalle I ;
- que le délai qui s'écoulera jusqu'à la date de la prochaine occurrence d'un évènement sera dans un intervalle I.

La vérification des propriétés ainsi exprimées se fait via l'exécution centralisée d'un automate temporisé.

Vérification distribuée

Même si les logiques temporelles semblent à première vue moins adaptées que les réseaux de Petri pour être évaluées de manière répartie, la vérification distribuée des propriétés logiques est abordée dans les travaux de Bauer et Falcone [BF12] et ceux de Sen et collab. [SVAR06].

Bauer et Falcone découpent chaque proposition logique en de multiples sous propositions, chacune vérifiable localement. Après avoir vérifié la partie qu'il était capable de calculer, un nœud envoie des propositions logiques aux autres nœuds afin qu'ils puissent continuer le calcul. Il faut remarquer qu'ils font l'hypothèse de liens synchrones et surtout que les messages envoyés peuvent être de taille relativement conséquente.

Sen et collab., de leur côté, ont créé leur propre logique. Ils utilisent ensuite des vecteurs (inspirés des horloges de Lamport) contenant un ensemble de variables globales et d'information sur le système. Puisqu'un nœud ne peut connaître instantanément l'état d'un autre nœud, il utilise les dernières informations qu'il a reçues pour calculer les propriétés désirées. Tout comme les travaux précédents, la taille des messages échangés peut être problématique.

D'autres approches cherchent à obtenir un moniteur réparti comme par exemple les travaux de Zhou et collab. [ZSLL09] cherchant à distribuer l'outil MAC [KVBAKLS99] décrit plus haut

et ceux de Jahanian et collab. [JRR94]. Ces derniers autorisent la vérification distribuée de propriétés exprimées comme une conjonction de simples propriétés logiques. Les travaux de Zhou et collab. quant à eux, proposent la distribution de propriétés exprimées en logique temporelle et se basent sur le langage de spécification MEDL.

Cependant, dans les deux cas, le formalisme proposé n'est pas aussi expressif que celui utilisé dans cette thèse — les réseaux de Petri A-temporisé — qui permet d'exprimer des comportements plus complexes et surtout distribués.

De plus, tous les travaux présentés ci-dessus font l'hypothèse d'une observation parfaite, hypothèse que nous ne faisons pas puisque nous considérons le cas d'observations bruitées avec comme conséquences des évènements non observés ou observés dans le désordre.

2.6.3. Réseaux de Petri

Les réseaux de Petri sont un des formalismes les plus répandus dans le domaine de la vérification. De même que les logiques temporelles, leur étude s'est généralisée à la vérification en ligne, éventuellement temps réel et distribuée.

Expressivité

Il existe de nombreux formalismes différents pour introduire la gestion du temps dans les réseaux de Petri. Les travaux de Boyer et Roux [BR08] décrivent et comparent ces différents formalismes.

En particulier, on peut associer, selon le formalisme, des intervalles de temps :

- aux places ; dans ce cas l'intervalle indique combien de temps un jeton peut rester sur la place en question.
- aux arcs allant des places aux transitions ; dans ce cas l'intervalle indique combien de temps un jeton peut rester sur la place en question s'il franchit la transition à l'extrémité de l'arc. Ceci est clairement une généralisation du cas précédent (il suffit de mettre le même intervalle de temps à chaque arc sortant de la place pour retomber dans le cas précèdent) ;
- aux transitions ; dans ce cas leur franchissement n'est plus atomique, mais au contraire peut prendre un certain temps.

À chacun de ces formalismes, on peut associer deux sémantiques. La sémantique forte demande à ce que si une transition t est franchissable jusqu'à un instant τ, alors elle doit être franchie au plus tard à la date τ (sauf si entre temps, le franchissement d'une autre transition a rendu t non franchissable). À contrario, la sémantique faible autorise une transition à passer d'un état franchissable à un état non franchissable en laissant le temps passer.

En combinant les sémantiques avec les trois options ci-dessus, on se retrouve avec six formalismes différents possible ! Le formalisme que nous avons utilisé dans cette thèse (temps associé aux arcs et sémantique faible) est le plus général dans le sens où il peut simuler tous les autres. [BR08]

Exécution simple d'un réseau de Petri

Un réseau de Petri permet de vérifier des propriétés distribuées assez facilement en franchissant des transitions lorsqu'un évènement associé est détecté. Si l'exécution du modèle se passe sans rencontrer d'erreurs, alors on peut considérer l'exécution du système correcte. Un tel exemple d'exécution peut se trouver dans les travaux de Zhu et Kordon [ZK10]. Cependant, contrairement à notre approche, cette exécution n'est ni distribuée ni ne permet de vérifier des propriétés temporelles.

Correspondance d'une observation avec une exécution de réseau de Petri

Une autre approche de supervision centralisée de systèmes distribuées, qu'ils soient temps réel [CJ05] ou asynchrones [FBHJ05 ; BHFJ03], sur la base des réseaux de Petri temporisés a aussi été considérée. Dans ces travaux, la principale difficulté consiste à trouver, après exécution, une exécution d'un réseau de Petri qui explique, à posteriori, une observation donnée. En effet, dans ces travaux, à chaque évènement peut correspondre plusieurs transitions. Les objectifs que nous nous fixons sont différents. D'abord nous faisons l'hypothèse d'une correspondance évènement/transition, ce qui rend trivial le problème considéré dans l'hypothèse d'une observation parfaite. Enfin, la plus grande différence réside dans le fait que nous nous intéressons à exécuter le réseau de Petri en ligne et en présence d'observations bruitées avec comme conséquences des évènements non observés ou observés dans le désordre.

Exécution d'un réseau de Petri de manière distribuée

Les réseaux de Petri distribuables, introduits dans les travaux de Hopkins [Hop91], s'apparentent davantage à notre problématique. L'idée consiste à répartir un réseau de Petri sur différents nœuds d'un système distribué en se basant sur la notion de *fonction de localisation.* Une fonction de localisation est une fonction qui indique sa position sur la réseau pour chaque place et chaque transition et vérifiant la propriété suivante :« Si une place p précède une transition t, c'est-à-dire si $p \in {}^\bullet t$, alors p et t doivent être situés sur le même nœud » ; cela autorise les franchissements de transition à se décider localement.

Leur problématique est sensiblement distincte de la nôtre dans le sens où ils ne cherchent pas à faire correspondre une exécution de réseau de Petrià une observation , mais simplement à franchir les transitions en respectant la sémantique. Par rapport à nos travaux, cela correspondrait à ne considérer que des réseaux de Petri ne contenant que des transitions logiques.

D'ailleurs nous nous sommes inspirés de la notion de fonction de localisation pour définir ce que nous avons appelé *bloc logique* et qui correspondent à des sous parties de réseau de Petri atomique (qui ne peuvent être répartis sur plusieurs nœuds).

2.6.4. Utilisation des moniteurs

Comme nous l'avons mentionné dans l'introduction, l'un des usages permis par la présence d'un moniteur est la mise en place de boucles de rétroactions. Pour faire simple, le superviseur utilise les connaissances qu'il a sur le système pour le modifier de manière adaptée. Nous allons décrire brièvement trois de ces usages.

Le premier cas concerne l'injection de fautes ; le principe est de tester le comportement d'un système lorsque l'environnement dans lequel il évolue ne respecte pas les spécifications. L'étude présentée dans [FA+03] montre justement l'intérêt d'un moniteur pour cette catégorie de test. En effet, dans ces travaux, les auteurs ont mis en place un mécanisme qui injecte les fautes quand le système est dans un état donné. Pour ce faire, ils sont obligés de surveiller le système en question pour connaître son état. Il serait peut-être abusif de parler de moniteur pour ces travaux, car l'objectif n'était pas tant la mise en place d'un superviseur généraliste que le test de scénarios d'injections de fautes simples. Cependant, pour tester des scénarios plus complexes et plus généraux, il pourrait être intéressant d'utiliser un véritable moniteur.

Un autre exemple d'usage d'un moniteur pour la mise en place de boucle de rétroaction est donnée par une approche d'ordonnancement dynamique de tâches critiques et de tâches non critiques sur un système multicœurs [KBPRR14 ; KBPRRV13]. Puisque l'ordonnancement temps réel est toujours extrêmement pessimiste, la plupart du temps, le processeur est sous-utilisé. Le principe est alors d'adopter une approche optimiste en lançant des tâches non critiques en parallèle et de superviser l'avancement de la tâche critique pour être sûr qu'elle sera capable de vérifier les contraintes temps réel auxquelles elle est soumise. Si ce n'est plus le cas, le moniteur tue les autre tâches afin de garantir le respect du délai associé à la tâche principale.

Dans un tout autre domaine, Chandra et Toueg, se sont intéressés aux possibilités offertes par les détecteurs de fautes. Dans leur modèle, un processus fautif est un processus qui n'interagit plus avec le reste du système, que ce soit dû à une problème logiciel, une panne d'origine matérielle ou une simple déconnexion. En particulier, ils montrent que l'existence d'un protocole détectant les processus fautifs suffit à offrir une solution au problème du consensus dans un réseau asynchrone [1]. En particulier, ils montrent même qu'une version plus faible de ce détecteur de fautes suffit [CT96 ; CHT96].

2.6.5. Conclusion

La mise en place d'un outils dédié à la supervision distribué, temps réel (souple) dans un contexte d'observation bruité n'as jamais été, à notre connaissance, explicitement considéré.

1. Nous rappelons au lecteur que le problème du consensus en environnement asynchrone à été montré impossible par Fisher, Lynch et Paterson [FLP85]

Chapitre III

Supervision d'un modèle évènementiel

> Et aussitôt elle criait tout bas :
> « Anne, ma sœur Anne, ne vois-tu rien venir ? »
> Et la sœur Anne répondait : « Je ne vois rien que le soleil qui poudroie, et l'herbe qui verdoie. »
>
> *Barbe bleue*
> Charle Perrault

Comme nous l'avons annoncé dans le chapitre précédent, nous allons détailler dans cette partie la mise en place d'un moniteur. En particulier une telle étude nécessite de décrire en détail chaque partie de notre moniteur. Pour rappel, un moniteur prend en entrée un modèle et l'anime ensuite à partir d'une observation.

Dans cette même partie, nous avons présenté différents modèles d'exécution dont la sémantique repose sur deux catégories de transitions distinctes : les transitions évènementielles correspondant aux évènements générés par les capteurs et les transitions logiques dont le franchissement est conditionné par l'état du modèle et qui permettent de vérifier certaines propriétés.

Nous nous concentrerons dans ce chapitre sur les réseaux de Petri constitués uniquement de transitions évènementielles. La généralisation avec les transitions logiques fera l'objet du chapitre IV.

Le formalisme utilisé pour modéliser autant le système que les propriétés associées est celui des réseaux de Petri A-temporisés ; il fera l'objet de la section 3.1. Dans un second temps, dans la section 3.2 nous présenterons de manière informelle l'approche générale. La sémantique de notre formalisme qui correspond à une généralisation de celle des réseaux de Petri A-temporisés sera décrite dans la section 3.3.

Ensuite, section 3.4, nous présenterons l'algorithme de notre superviseur uniquement dans le cas où les transitions sont toutes évènementielles. Enfin, nous discuterons, section 3.5, la tolérance aux fautes de notre système et comment des politiques de redondance permettent de l'améliorer.

3.1. Modèles et définitions

Le formalisme utilisé dans cette section généralise la notion de réseau de Petri A-temporisé. Dans cette partie nous allons définir le modèle de ces derniers ainsi que leur sémantique. Cependant, nous donnons ces informations à titre informatif et pédagogique. En effet, nous utiliserons par la suite, dans la section 3.3, une définition et une sémantique légèrement différente.

3.1.1. Définition d'un réseau de Petri A-temporisé

Commençons par la définition formelle avant de donner un exemple. Dans la définition suivante, 2^E représente l'ensemble des parties (des sous-ensembles) d'un ensemble E.

Définition 2 (Réseau de Petri A-temporisé).

Un réseau de Petri A-temporisé est un tuple $(P, T, pre, post, p_0, I)$ où P est l'ensemble des places, T, l'ensemble des transitions, pre et $post$ deux fonctions de T à valeur dans 2^P, p_0 représente l'état initial et I est une fonction qui à chaque couple $(p,t) \in P \times T$ vérifiant $p \in pre(t)$ associe un intervalle de $\mathbb{R}^+$.

En guise d'illustration considérons l'exemple de la figure III.1. On a $P = \{p_1, p_2, p_3\}$, $T = \{t_2, t_3\}$, $pre(t_2) = pre(t_3) = \{p_1\}$, $post(t_2) = \{p2\}$ et $post(t_3) = \{p_2, p_3\}$. La fonction I est définie pour deux couples : $I(p_1, t_2) = [3, 5]$ et $I(p_1, t_3) = [2, +\infty[$. Dans la suite de ce document on utilisera comme notation pour toute transition t et pour toute place p :

- ${}^\bullet t = pre(t)$ et $t^\bullet = post(t)$
- $p^\bullet = \{t \mid p \in {}^\bullet t\}$ et ${}^\bullet p = \{t \mid p \in t^\bullet\}$
- $I(p,t) = I_{p,t}$

Intuitivement, un réseau de Petri est un graphe orienté ; l'ensemble des successeurs d'un nœud n (un nœud pouvant être une place où une transition) sera marqué « $n^\bullet$ », l'ensemble des prédécesseurs sera marqué « ${}^\bullet n$ ».

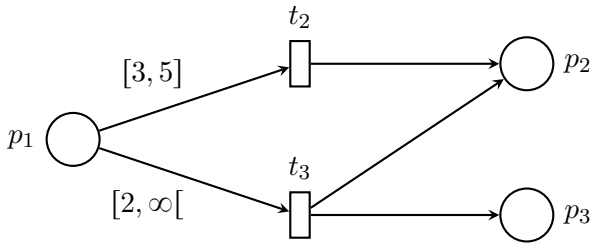


Figure III.1.: Un réseau de Petri

Par convention, si pour un couple (p,t) où p est une place et t une transition, si la valeur de l'intervalle $I_{p,t}$ n'est pas indiquée, on considérera par défaut que $I_{p,t} = [0, \infty[$. Comme

la section suivante parlant de la sémantique le montrera, ce choix revient à ne mettre aucune condition sur la durée de vie d'un jeton sur cette place.

Définition 3 (Marquage ou état).

Un marquage est une fonction qui associe à certaines places de P un jeton. Étant donné que l'on considère des réseaux de Petri temporisés, la date d'apparition de chaque jeton lui sera associé. Soit M un marquage, la présence d'un jeton associé à une date τ sera définie par $M(p) = \tau$; dans le cas où il n'y a aucun jeton sur la place, on notera alors $M(p) = \bot$.

Remarquons qu'avec cette définition, certaines places peuvent ne pas contenir de jeton, mais surtout qu'il ne peut pas y avoir plus d'un jeton par place. Cette condition qui peut sembler un peu forte disparaîtra avec notre modèle.

3.1.2. Sémantique d'un réseau de Petri A-temporisé

La sémantique associée à cette famille de réseaux de Petri est une simple généralisation de la sémantique des réseaux de Petri non temporisés.

Définition 4 (Sémantique d'un réseau de Petri $\mathcal{A}$-temporisé).

Une sémantique d'un réseau de Petri $\mathcal{P}$ est un graphe dont les sommets sont des états de $\mathcal{P}$ et dont les arêtes correspondent à des transitions datées ; une transition datée étant un couple (t, τ) où t est une transition et τ une date.

Dans la suite étant donné que l'on ne considérera qu'une sémantique, on se permettra dans la suite de parler incorrectement de *la* sémantique plutôt que de dire *une* sémantique.

Définition 5 (Exécution d'un réseau de Petri $\mathcal{A}$-temporisé).

Une exécution d'un réseau de Petri est une suite de transitions datées. Une exécution est dite correcte, vis-à-vis d'une sémantique donnée, si la suite de transitions datées correspond à une chemin dans le graphe de la sémantique.

Définition 6 (Transitions franchissables).

On dit qu'une transition datée (t, τ) est franchissable dans l'état σ si :

- pour tout p de ${}^\bullet t$ on a $M(p) \neq \bot$;
- pour tout p de ${}^\bullet t$ on a $\tau - M(p) \in I(p, t)$.
- pour tout p de $t^\bullet$ on a $M(p) = \bot$;

Dit de manière plus informelle, une transition est dite franchissable si :

1) il existe un jeton dans chaque place la précédant ;
2) la date d'apparition de ces jetons est compatible selon I avec la date τ du franchissement ;
3) les places qui succèdent à t sont bien vides.

Considérons par exemple la figure III.2. Supposons que dans la place p_1 un jeton soit apparu à la date 5 et que dans la place p_2 un autre jeton soit apparu à la date 3. Le marquage associé est définit par $M(p_1) = 5$, $M(p_2) = 3$ $M(p_3) = \bot$. Dans ce cas, la transition sera franchissable à la date 8 car :

- le jeton de la place p_1 y est resté $8-5 = 3$ unités de temps et la valeur 3 est bien comprise dans l'intervalle $I(p_1, t) = [3, 5]$;
- le jeton de la place p_2 y est resté $8-3 = 5$ unités de temps et la valeur 5 est bien comprise dans l'intervalle $I(p_2, t) = [2, \infty[$;
- il n'y a pas de jeton dans la place p_3.

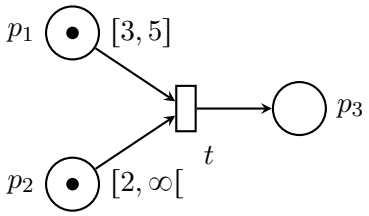


Figure III.2.: Transition franchissable

Par contre, la transition n'était pas franchissable à la date 7, car si dans le cas du jeton de la place p_2 on a bien $7 - 3 \in [2, \infty[$, dans le cas de la place p_1 on obtient $7 - 5 = 2 \notin [3, 5]$.

Il est important de bien faire la différence entre date et durée. Les évènements sont associés à des dates ; les intervalles de temps associés aux arcs correspondent à des durées. Ainsi, on fait la différence entre la date d'apparition d'un jeton et sa date de disparition pour vérifier que la durée calculée est bien incluse dans l'intervalle indiqué sur l'arc.

Définition 7 (Arêtes du graphe de la sémantique).

Soient P un réseau de Petri et σ et σ' deux états de P. Alors, dans le graphe de la sémantique il existe une arête entre σ et σ' représentée par la transition datée (t, τ) si :

- (t, τ) est franchissable dans σ ;
- Pour tout p n'appartenant pas à ${}^\bullet t \cup t^\bullet$, $M(p) = M'(p)$.
- Si $p \in {}^\bullet t$ alors $M'(p) = \varnothing$
- Si $p \in t^\bullet$ alors $M'(p) = \tau$

De manière informelle, le second point signifie que les jetons qui ne sont pas dans les places connectées à t sont invariants par le franchissement de la transition t ; le second point indique que les jetons situés sur les places qui précèdent t doivent être supprimés ; le troisième point, enfin, indique que des jetons sont créés dans les places qui succèdent t, et ces jetons sont indexés par la date du franchissement τ.

3.1.3. Modèle d'erreurs

Si l'on désire maintenant utiliser une telle sémantique pour notre modèle à superviser, il « suffit » de recevoir une liste d'évènements et de l'exécuter selon la sémantique présentée dans la partie précédente. Bien évidemment, il se peut que des erreurs soient rencontrées durant les tentatives d'exécution. Avant de parler de ces erreurs, il nous faut définir une nouvelle notion :

Définition 8 (Transitions pseudo-franchissables).

On dit qu'une transition datée (t, τ) est pseudo-franchissable dans l'état σ si :

- pour tout p de ${}^\bullet t$ on a $M(p) \neq \perp$;
- pour tout p de $t^\bullet$ on a $M(p) = \perp$.

Pour faire simple, une transition pseudo-franchissable dans un état d'un réseau de Petri serait franchissable si l'on considérait le réseau de Petri classique (non temporisé) associé en faisant abstraction des dates.

Cette notion permet de clarifier les trois types différents d'erreur qui peuvent être rencontrées durant l'exécution d'un réseau de Petri A-temporisé : 1) les évènements non franchissables, 2) les erreurs temporelles et 3) les jetons morts. Ces erreurs sont déclenchées lors de l'exécution du modèle suite à la réception d'un évènement.

Définition 9 (Erreur d'évènements non franchissables).

Une erreur d'évènements non franchissables se produit lorsque l'évènement reçu n'est pas pseudo-franchissable dans l'état courant de notre modèle.

Cette erreur est la plus problématique. Elle peut être due à plusieurs raisons comme par l'exemple l'omission d'un message ou simplement un comportement du système incompatible avec le modèle choisi. Cette erreur est gênante car il suffit d'un évènement manqué pour bloquer l'intégralité de l'exécution et rendre tous les futurs évènements non franchissables !

Définition 10 (Erreurs temporelles).

Une erreur temporelle se produit lorsque la transition est pseudo-franchissable mais n'est pas franchissable.

Une telle erreur signifie qu'un jeton est tiré soit trop tard, soit trop tôt. Cependant cela n'empêche pas l'exécution des prochains évènements.

Le cas de la présence de jeton mort est plus subtile et nécessite d'enrichir légèrement notre modèle par la présence d'une horloge globale. Pour définir cette erreur, cette horloge ne doit pas forcément être accessible mais simplement exister. Considérons de nouveau le réseau de Petri de la figure III.2 et supposons qu'un jeton soit arrivé dans la place p_1 à la date 1, dans ce cas pour ne pas avoir d'erreur temporelle il faut que la transition t soit franchie entre les dates $[4, 6]$; si l'horloge indique 7, alors on sait que ce jeton ne pourra jamais franchir t sans provoquer une erreur temporelle. D'aucun pourrait penser que ce cas est traité dans les deux premiers type d'erreur, mais ce serait une erreur : tant qu'aucun autre évènement n'est reçu, il n'y a pas de franchissement impossible ni de franchissement temporellement fautif.

Définition 11 (Erreur de jeton mort).

Soient p une place contenant un jeton daté de la date τ et T la date indiquée par l'horloge globale. Si pour toute transition t de $p^\bullet$ on a :

$$T - \tau > \sup(I_{p,t}) \tag{III.1}$$

alors on dit que ce jeton est mort.

3.2. Approche générale : exécution asynchrone d'un réseau de Petri

Cette partie a pour objectif de présenter rapidement et sans entrer dans les détails notre approche pour exécuter en temps réel un réseau de Petri à partir d'une observation imparfaite. En particulier elle introduira les notions de jetons négatifs et d'identifiants de jetons.

3.2.1. Des jetons positifs et négatifs

Considérons le réseau de Petri de la figure III.3 et la réception des trois évènements $(t_1, 10)$, $(t_2, 15)$ et $(t_3, 21)$. L'idée est de comprendre pourquoi la sémantique classique des réseaux de Petri A-temporisés n'est pas suffisante pour vérifier la validité d'une observation à l'exécution.

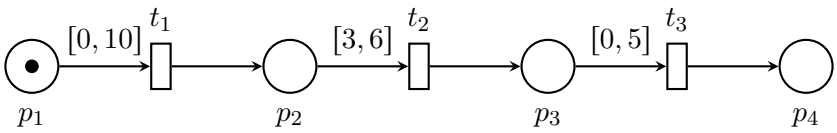


Figure III.3.: Un réseau de Petri

Une observation parfaite Supposons que le moniteur reçoit dans cet 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 $15 - 10$ est bien dans l'intervalle $[3, 6]$, 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 le temps passé par le jeton de la place p_3 donné par $21 - 15$, n'est pas dans l'intervalle de temps considéré $[0, 5]$. Le moniteur tire quand même la transition car l'évènement a bel et bien eu lieu, mais soulèvera une erreur temporelle car l'évènement correspondant à t_3 est arrivé trop tard.

Une observation imparfaite 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.

Dans notre approche, 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 s'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 $t^\bullet$.

Une exécution avec des jetons négatifs Notre approche consiste à franchir une transition dès la réception de l'évènement associé en anticipant la suppression du jeton des places de ${}^\bullet t$; cette anticipation se fait via l'ajout de jetons négatifs. Le résultat du franchissement du t_2 est représenté sur la figure III.4. Les jetons négatifs sont représentés en blanc et les jetons positifs en noir.

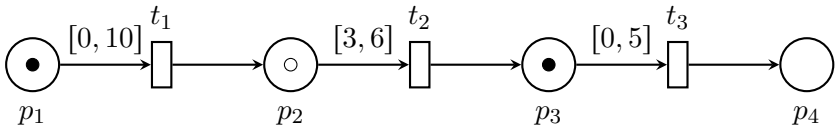


Figure III.4.: Après le franchissement de t_2

Comme nous pouvons le voir figure III.5, Le franchissement de t_3 ajoute un jeton négatif dans la place p_2 et un jeton positif dans la place p_3. La suppression d'un jeton dépend de la présence de son *alter ego* de signe opposé dans la même place. Ainsi, on peut supprimer les deux jetons de la place p_3.

Pour savoir le temps passé par le jeton positif dans la place p_2, il suffit de comparer les dates associées aux deux jetons, positif et négatif. Le jeton positif a été ajouté lors de la réception de l'évènement $(t_2, 15)$; de même le jeton négatif a été ajouté lors de la réception de l'évènement $(t_3, 21)$. Ainsi, le jeton est resté $21 - 15$, c'est-à-dire 6 unités de temps sur la place p_2. Or, l'intervalle de temps spécifié étant $I(p_2, t_3) = [0, 5]$, une erreur temporelle sera levée. Enfin, les deux jetons peuvent être supprimés.

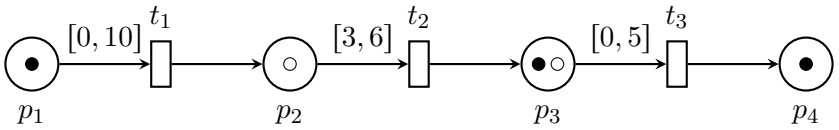


Figure III.5.: Après le franchissement de t_3

Des erreurs temporelles et des erreurs d'omissions La figure III.5 nous montre un état dans lesquelles les jetons permettent de détecter trois erreurs de deux types : une erreur temporelle place p_3, des jetons morts place p_1 et p_2.

- **Des erreurs temporelles** Tout d'abord, les deux jetons de la place p_3 induisent, comme nous venons de le voir, un erreur temporelle ; les spécifications temporelles du modèle indiqué par l'intervalle $I(p_3, T_3) = [0, 5]$ ne sont pas respectées.
- **Des jetons morts** Le jeton de la place p_1 va mourir et enfin la présence d'un jeton négatif sur la place p_2 pose problème. Une erreur d'évènement non franchissable se traduit maintenant par un jeton négatif mort (c'est-à-dire un jeton négatifs dont l'*alter ego* positif n'est pas arrivé dans les temps). Admettons que par la suite, la version de signe opposé de ce jeton ne soit jamais reçue (i.e. si le jeton positif n'est jamais reçu ; trois scénarios indistinguables peuvent être à l'origine de cette absence d'observation :
  1) l'évènement n'a jamais eu et n'aura jamais lieu ;
  2) l'évènement a eu lieu dans les temps mais le message correspondant s'est perdu ;
  3) l'évènement a eu lien trop tard ou trop tôt mais le message correspondant s'est perdu.

  Comme nous pouvons le voir, dans deux cas sur trois, la présence d'un jeton mort correspond à une erreur dans le système. Cette non observation (correspondant à deux jetons morts) sera qualifiée d'erreur d'omission. Si par manque d'information notre moniteur peut déclencher une erreur inexistante (faux positif), on remarque que l'on s'interdit la non détection d'une erreur d'exécution qui a effectivement eu lieu (faux négatif).

En résumé, la principale différence entre notre approche et celle basée sur la sémantique classique des réseaux de Petri est la suivante : alors que dans la sémantique classique la validité

du franchissement est faite à priori, imposant de recevoir tous les évènements et dans le bon ordre, notre sémantique franchit systématiquement la transition associée à l'évènement reçu et vérifie à postériori que ce franchissement était correct.

3.2.2. Jetons et identifiants

Dans le cadre des réseaux de Petri temporisé, la sémantique classique impose une condition relativement forte : il ne peut pas y avoir plus de deux jetons par places. Ceci est dû au fait que le fait de franchir une transition ne nous permet pas de savoir quel jeton tirer, si deux d'entre eux se trouvent dans la même place.

Le problème d'une telle condition tient du fait que l'on s'autorise à avoir des exécutions partielles qui peuvent mener à avoir au moins deux jetons dans une même place, l'un n'ayant pas été supprimé à la suite d'un évènement manquant.

Définition 12 (Jetons).

Un jeton est un couple (s, id) où s représente le signe et peut être soit « $+$ » soit « $-$ » 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 III.6, deux jetons positifs sont dans une place. Supposons que le jeton d'identifiant « 1 » soit 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.

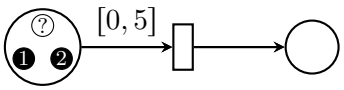
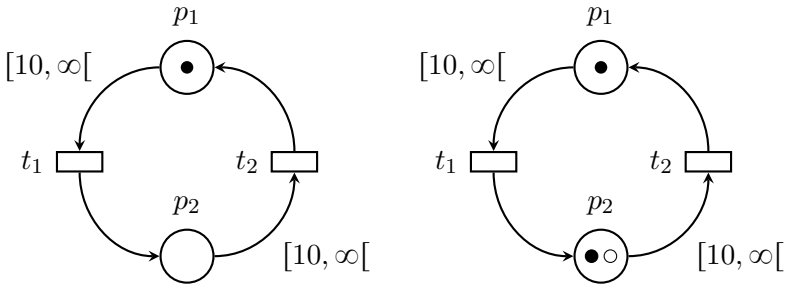


Figure III.6.: Jetons et identifiants

3.2.3. Évènements

Ce genre d'ambiguïté peut arriver régulièrement à partir du moment où le réseau de Petri contient des boucles. Prenons l'exemple du réseau de Petri de la figure III.7 et la suite d'évènements $(t_1, 0)$, $(t_2, 5)$, $(t_1, 10)$ et $(t_2, 15)$. Considérons les deux scénarios suivants. Dans le premier cas, deux évènements ont lieu et sont observés : $(t_1, 0)$ et $(t_2, 15)$. Dans le second cas, quatre évènements ont lieu $(t_1, 0)$, $(t_2, 5)$, $(t_1, 10)$ et $(t_2, 15)$, mais cette fois, seul le premier et le dernier sont reçus par le moniteur. Il est aisé de voir que du point de vue du moniteur

ces deux scénarios sont indistinguables. Cependant, le premier correspond à un scénario fautif quand le second correspond à un scénario correct. L'ajout d'identifiants permet de différencier ces deux scénarios.



(a) Début de l'exécution (b) Cas d'ambiguïté

Figure III.7.: Jeton et ambiguïté

Pour cela on redéfinit la notion d'évènement : un évènement est un triplet (t, τ, id) où t est une transition, τ une date et id une fonction qui à chaque place de ${}^\bullet t$ et de $t^\bullet$ renvoie un identifiant. En prenant en compte cette généralisation des évènements on peut récrire les deux scénarios. Le premier devenant:

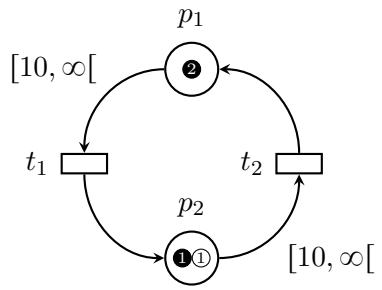
- $(t_1, 0, (p_1 \to 0, p_2 \to 1))$
- $(t_2, 15, (p_2 \to 1, p_1 \to 2))$

La syntaxe signifie que le franchissement de t_1 ajoutera un jeton d'identifiant 0 dans p_1 et un jeton d'identifiant 1 dans p_2. Quand au second scénario, on obtient

- $(t_1, 0, (p_1 \to 0, p_2 \to 1))$
- $(t_2, 5, (p_2 \to 1, p_1 \to 2))$
- $(t_1, 10, (p_1 \to 2, p_2 \to 3))$
- $(t_2, 15, (p_2 \to 3, p_1 \to 4))$

Cette fois-ci, comme montré dans la figure III.8(a), même en perdant les deux évènements $(t_2, 5)$ et $(t_1, 10)$, les deux exécutions seront différentiables et seule la seconde déclenchera une erreur. Dans le premier cas, les deux jetons de la place p_2 ont le même identifiant et donc peuvent se simplifier, quand au second cas, les deux identifiants étant différents, cela signifie que les évènements correspondant n'ont pas été encore reçus — et ne le seront peut-être jamais.

Avant de continuer, donnons une définition plus formelle de cette notion.



(a) p_2 est simplifiable (b) p_2 n'est pas simplifiable

Figure III.8.: Jeton et ambiguïté

Définition 13 (Évènement).

Un évènement est 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).

Définition 14 (Séquence d'évènements).

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 $1 \leqslant i < j, \leqslant n$ et pour toutes places p l'égalité $M_i(p) = M_j(p)$ implique :

1) $t_i \in {}^\bullet p$;
2) $t_j \in p^\bullet$;
3) si $k \neq i$ et $k \neq j$ alors $M_k(p) \neq M_i(p)$

La dernière condition affirme l'unicité des identifiants des jetons pour une place donnée. Pour que les identifiants permettent de différencier les jetons d'une même place il faut que pour une place et un identifiant donnés, deux jetons seulement portent cet identifiant : 1) l'un de ces jetons doit être positif, 2) l'un doit être négatif et 3) tous les autres jetons apparaissant sur ces places doivent posséder un identifiant différent.

Bien évidemment une séquence d'évènements correctes est théorique. Notre observation pourra contenir des séquences partielles et désordonnées.

3.2.4. De la légitimité de l'usage d'identifiant

Cette information est-elle nécessaire ? D'aucun peut légitimement se poser la question de l'implémentabilité d'un système avec identifiant. En effet, cette information associée à chaque

jeton peut sembler relativement forte et faire appel à un tel mécanisme peut ressembler à ce qu'un esprit chagrin pourrait appeler de la « triche ». Cependant, il faut bien comprendre que cette information est nécessaire à la supervision.

Considérons un exemple simple. Une machine ne peut être utilisée que par une personne à la fois. On veut vérifier qu'elle reste au moins dix minutes devant. Si un homme brun arrive à 13h et repart à 13h05 et qu'une femme rousse arrive à 13h06 et repart à 13h11, on remarque qu'aucune de ces deux personnes n'est restée plus de dix minutes devant la machine.

En cas de problème d'observation notre superviseur peut observer l'homme arriver à 13h et la femme repartir à 13h10. S'il n'est pas capable de différencier les deux individus, il ne détectera aucune erreur. Un lecteur un peu attentif aura compris que l'on parle du même exemple que dans la partie précédente. Ce qui est important est surtout de comprendre que le problème ne dépend pas de notre modèle mais est un problème général de supervision. Si les observations sont trop peu précises, peu de propriétés pourront être supervisées.

Existe-il des alternatives aux identifiants ? Une première remarque consisterait à remarquer que l'on pourrait résoudre ce problème avec moins d'information. Ce qui est important est d'avoir une relation d'équivalence entre jeton pour identifier le jeton positif avec le négatif. Nous nous sommes concentrés sur la relation d'égalité car c'est la façon la plus simple de résoudre un tel problème, problème qui est de toutes façons orthogonal aux autres problématiques rencontrées dans cette thèse. Choisir une autre solution ne changera rien à nos algorithmes et protocoles.

Comment générer les identifiants ? Cette discussion sera faites dans la partie 5.4. L'idée étant d'utiliser soit l'identifiant associé naturellement à une requête sur un serveur (par exemple, l'identifiant du robot), soit d'utiliser un compteur qui permet d'identifier le numéro de passage dans la boucle.

3.3. Sémantique

L'objectif de cette partie est de définir la sémantique de notre formalisme. Dans une première partie, section 3.3.1 nous introduirons la notion d'état. Une sémantique étant un graphe dont les nœuds sont des états et les arêtes des évènements (cf. définitions 4), nous définirons la notion d'état ; la notion d'évènement ayant été introduite dans la partie précédente (cf. définition 13) on introduira alors, section 3.3.2, la sémantique dans le cas où seules les transitions évènementielles sont considérées.

3.3.1. État

Comme annoncé précédemment, nous utilisons notre propre version des réseaux de Petri. Cette partie ne considérant pas les transitions logiques ne donne qu'une définition temporaire qui sera généralisée dans le chapitre suivant. La principale différence avec la définition 2 de la section 3.1 est la considération de deux types de transitions nommé T_e et T_ℓ pour indiquer

respectivement les ensembles de transitions évènementielles et logiques. Pour l'instant, dans cette partie, nous imposons un ensemble T_ℓ vide.

Définition 15 (Réseau de Petri de supervision simple).

Un réseau de Petri de supervision est un tuple $(P, T_e, T_\ell, pre, post, p_0, I)$ où P est l'ensemble des places, T_e, l'ensemble des transitions évènementielles, $T_\ell = \varnothing$, l'ensemble des transitions logiques, pre et $post$ deux fonctions de $T = T_e \cup T_\ell$ à valeur dans 2^P, p_0 représente l'état initial et I est une fonction qui à chaque couple $(p, t) \in P \times T$ vérifiant $p \in pre(t)$ associe un intervalle de $\mathbb{R}^+$.

Nous avons vu dans la partie précédente qu'il y avait deux types possibles d'erreurs : les erreurs temporelles et les erreurs d'observation. Les premières correspondent à une spécification temporelle non respectée ; les secondes à un évènement qui n'a pas eu lieu ou qui n'a pas été reçu.

La sémantique étant définie de manière statique, c'est-à-dire sans faire usage d'horloge globale, seules les erreurs temporelles pourront être adressées par la sémantique. S'il reste des jetons dans les places à la fin de l'exécution, cela signifie que ces jetons sont morts ; cependant la mort d'un jeton ne peut dans ce cas n'être découverte qu'après l'exécution et non pendant, car leur définition (définition 11) nécessite une horloge globale.

En cas d'erreur, l'information associée doit être stockée en vue de comprendre à postériori ce qui s'est passé. Pour cela on définit la représentation d'erreur de type temporel qui sera stockée lorsque une erreur temporelle sera détectée.

Définition 16 (Représentation d'une erreur temporelle).

Une erreur temporelle est représentée par un tuple $(p, t, \tau_+, \tau_-, id)$. Une telle erreur signifie que dans la place p, il y avait deux jetons de même identifiant id, le jeton positif de date τ_+, et le jeton négatif de date τ_- et apparu via le franchissement de la transition t tel que :

$$\tau_- - \tau_+ \notin I(p, t)$$

Définition 17 (État d'un réseau de Petri de supervision).

L'état d'un réseau de Petri de supervision est un couple (M, E) avec M un marquage et E un ensemble de représentation d'erreurs temporelles.

3.3.2. Sémantique

Plutôt que de définir directement un graphe, nous allons définir les séquences d'évènements acceptées. Puisqu'un évènement daté correspond à une arête d'un graphe de sémantique, une

séquence d'évènements correspond à un chemin. Le graphe sera simplement celui défini comme étant le plus petit graphe contenant tous ces chemins.

Définition 18 (Observation supervisable d'évènements).

Soit $\mathcal{E} = e_1 \dots e_n$ une séquence correcte d'évènements. On appelle observation supervisable d'évènements n'importe quelle séquence de la forme $e_{\phi(1)} \dots e_{\phi(k)}$ où ϕ^{-1} est une injection de $[\![1, k]\!]$ dans $[\![1, n]\!]$.

De manière moins formelle la définition précédente veut simplement dire que pour obtenir une observation supervisable d'évènements, il suffit de prendre une séquence d'évènement (définition 14), d'en enlever certains évènements et de changer l'ordre de ceux qui restent. Par exemple e_2, e_5, e_1 est une observation supervisable de e_1, e_2, e_3, e_4, e_5.

Il nous reste alors à définir les nœuds connectés par les évènements. Pour cela on définit d'abord l'état initial puis dans un second temps l'état obtenu par le franchissement d'un évènement.

Définition 19 (État initial).

L'état initial est l'état $(M_0, \varnothing)$ où M_0 correspond au marquage initiale : un jeton dans la place p_0 avec l'identifiant par défaut.

Nous allons maintenant expliciter le changement d'état résultant de la réception d'un événement (t, τ, id). Pour cela nous allons considérer séparément trois catégories de places : les places de ${}^\bullet t$, les places de $t^\bullet$ et les autres. Certaines places se verront ajouter un jeton positif, d'autres un jeton négatif, certaines les deux et enfin certaines aucun. Définissons d'abord le format des jetons. Dans la sémantique classique des réseaux de Petri temporisés, un jeton est une simple date. Cependant, du simple fait de l'ajout des jetons négatifs, il faut rajouter l'information du signe. L'ambiguïté des jetons nécessite en plus l'ajout des identifiants. Cependant ces informations ne sont pas encore suffisantes en ce qui concerne la vérification des propriétés temporelles ; en effet, il faut savoir de quelle transition vient le jeton négatif : suivant la transition sortante, l'intervalle à considérer n'est effectivement pas le même. Un jeton positif sera donc de la forme $(+, id, \tau)$; un jeton négatif sera de la forme $(-, id, \tau, t)$ où t correspond à la transition sortante.

Définition 20 (Jetons positifs et négatifs).

Un évènement (t, τ, id) produit un jeton positif $(+, id, \tau)$ qui sera placé dans les places de $t^\bullet$ et un jeton négatif, $(-, id, \tau, t)$, que sera placé dans les places de ${}^\bullet t$.

Pour des raisons de symétries et de concisions dans la description des algorithmes, on utilisera parfois la convention selon laquelle un jeton est toujours de la forme $(signe, id, \tau, t)$ même si, dans le cas où le signe est positif, il n'est pas nécessaire de préciser la transition qui a été franchie.

Pour en revenir à la définition de la sémantique, l'état atteint suite au franchissement d'une transition est celui obtenu en ayant modifié chaque place suivant les indications ci-dessous.

Ajout d'un jeton positif Soit q une place de $t^\bullet$. Lors de l'ajout du jeton positif il faut considérer deux cas, suivant qu'il existe ou qu'il n'existe pas dans la place un jeton négatif possédant le même identifiant. Dans le premier cas il faut supprimer le jeton déjà présent et vérifier que les dates correspondent aux spécifications ; dans le second, il suffit d'ajouter un jeton positif.

- S'il existe un jeton j_m tel que $j_m = (-, id(q), \tau_m, t_m) \in M(q)$, alors le nouveau marquage de la place est donné par $M'(q) = M(q)\backslash\{j_m\}$. Si de plus $\tau_m - \tau \notin I(q, t_m)$ alors, on rajoute une erreur à l'ensemble des erreurs en posant $E' = E \cup \{(p, \tau, \tau_m)\}$; sinon on a simplement $E' = E$.
- Si par contre un tel jeton j_m n'existe pas, on peut alors ajouter le nouveau jeton dans la place et on obtient $M'(q) = M(q) \cup \{(-, id(q), \tau)\}$ et $E' = E$.

Ajout d'un jeton négatif Soit q une place de ${}^\bullet t$. Lors de l'ajout du jeton négatif il faut, comme dans le cas précédant, considérer deux cas, suivant qu'il existe ou qu'il n'existe pas dans la place un jeton positif possédant le même identifiant. Dans le premier cas il faut supprimer le jeton déjà présent et vérifier que les dates correspondent aux spécifications ; dans le second, il suffit d'ajouter un jeton négatif.

- S'il existe un jeton positif j_p tel que $j_p = (+, id(q), \tau_p) \in M(q)$, alors le nouveau marquage de la place est donné par $M'(q) = M(q)\backslash\{j_p\}$. Si de plus $\tau - \tau_p \notin I(q, t)$ alors on rajoute une erreur à l'ensemble des erreurs en posant $E' = E \cup \{(p, \tau_p, \tau, -)\}$; sinon on a simplement $E' = E$.
- Si par contre une tel jeton j_p n'existe pas on obtient $M'(q) = M(q) \cup \{(+, id(q), \tau, t)\}$ et $E' = E$.

Autre places Pour tout q n'appartenant pas à ${}^\bullet t \cup t^\bullet$, on obtient simplement $M'(q) = M(q)$ et $E' = E$.

3.4. Algorithme et protocole : une approche distribuée

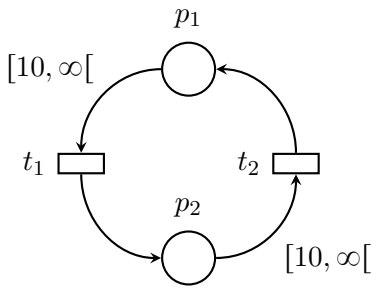
Si la sémantique permet de spécifier comment doit s'exécuter le réseau de Petri, son approche reste fondamentalement centralisée dans le sens où pour l'instant, l'état est une donnée atomique. Nous allons introduire dans cette partie une approche qui permet d'exécuter notre sémantique de manière décentralisée. Par la suite la notation « $p \quad ! \quad m$ » signifie l'envoi d'un message m au processus p.

3.4.1. Distribution du moniteur

L'idée de la distribution est de remarquer que le marquage est naturellement réparti sur les places et que l'état d'une place ne change que lorsqu'une transition $t \in {}^\bullet p \cup p^\bullet$ reçoit un événement. Il nous suffit alors d'associer à chaque place et à chaque transition un processus unique. Par la suite on identifiera les places et les transitions avec les processus associés ; par exemple, au lieu de dire que le processus associé à la place p a reçu un message m, on dira simplement que la place p a reçu le message m. L'algorithme global deviendra pour chaque réception d'évènement une succession de trois étapes dont les deux premières sont simplement des étapes de communication :

1) les transitions reçoivent des évènements ;
2) les transitions envoient des jetons positifs ou négatifs aux places ;
3) les places comparent les jetons, les simplifient et détectent les erreurs.

Le graphe de communication associé est représenté figure III.9(b). Lorsqu'un nouvel évènement arrive, par exemple e_1, e_2 ou e_3, le message est envoyé à la transition associée. Dans le superviseur, les transitions sont donc les premiers processus qui reçoivent les messages associés aux évènements envoyés par les capteurs. Les transitions n'ont plus qu'à créer les jetons à envoyer aux places. Ainsi si dans un réseau de Petri une flèche peut être dirigée d'une place vers une transition, dans notre superviseur une place n'envoie aucun message (si ce n'est des messages d'erreur à destination d'un opérateur extérieur au moniteur) et les seules communications entre une transition et une place se font de la première vers la dernière.



(a) Le réseau de Petri (b) Le graphe des communications

Figure III.9.: Réseau de Petri et communication

3.4.2. Procédure associée aux transitions évènementielles

Les transitions correspondent aux processus qui s'occupent de la gestion des évènements et la création des jetons. Lors de la génération de l'évènement (t, τ, id), cet évènement doit être envoyé à la transition t qui s'occupera de générer les jetons et de les envoyer aux différentes places. Le format des jetons est celui défini dans la définition 20 de la partie 3.3.2.

L'algorithme associé à une transition évènementielle t est donné dans l'algorithme 1 et se résume en trois phases : 1) un évènement est reçu (ligne 2), 2) le jeton négatif associé à l'évène-

ment est envoyé aux places de ${}^\bullet t$ (lignes 3 à 5) et enfin 3) le jeton positif associé à l'évènement est envoyé aux places de $t^\bullet$ (lignes 6 à 8).

```
1:  procédure Transition(t, Avant, Après)
2:      Quand un évènement e = (t, τ, id) est reçu
3:          Pour p ∈ Après faire
4:              p ! (+, id(p), τ, t)
5:          fin Pour
6:          Pour p ∈ Avant faire
7:              p ! (−, id(p), τ, t)
8:          fin Pour
9:      fin Quand
10: fin procédure
```

Algorithme 1: Procédure exécutée par une transition évènementielle t

3.4.3. Procédure associée aux places évènementielles

L'algorithme associé aux places (algorithme 2) est légèrement plus complexe et doit gérer deux catégories de messages. Le fonctionnement d'une place consiste à recevoir et à gérer des jetons ; cependant, contrairement à la sémantique présentée précédemment, il nous faut gérer le cas des jetons morts. Pour cela, à chaque fois qu'un nouveau jeton est reçu, un minuteur est lancé. L'algorithme associé au minuteur sera explicité par la suite. Ce minuteur nous permet de ne considérer que deux cas : soit la version négative du jeton sera reçue dans les temps (avant que le minuteur ne se termine), soit elle ne sera jamais reçue.

L'ensemble des jetons reçus et non morts est stocké dans l'ensemble Mem. Dans le cas où le minuteur se termine avant la réception du jeton correspondant, un message $timeout$ est reçu par la place, ce qui lui permet de supprimer ce dernier de l'ensemble Mem.

L'algorithme reste fidèle à la sémantique détaillée section 3.3.2, la seul différence étant sa capacité à détecter les jetons morts. Si aucun jeton n'est reçu, alors on ne fait rien. Si un jeton est reçu (lignes 2 à 13 de l'algorithme 2), on regarde si un jeton de même identifiant mais de signe opposé a déjà été reçu ; si la réponse est non, on rajoute le jeton et on lance un nouveau minuteur (lignes 10 et 11) ; si la réponse est oui, on annule le minuteur associé à l'autre jeton, on compare les deux jetons et éventuellement on déclenche une erreur. Enfin si un minuteur arrive à son terme, cela veut dire qu'un jeton mort vient d'être détecté : on peut donc le supprimer et déclencher une erreur (lignes à 14 à 17).

3.4.4. Procédure associée aux minuteurs évènementiels

L'usage du minuteur permet de savoir si la contrepartie négative d'un jeton sera reçue ou non. En effet, à partir du moment où un jeton n'est pas reçu avant la date limite, on fait l'hypothèse que ce jeton ne sera jamais reçu. Si jamais l'hypothèse s'avère fausse, les conséquences restent faibles. En effet, dans le cas où le jeton négatif est reçu après avoir décrété la mort du jeton

```
1:  procédure Place(I)
2:      Quand un jeton j = (signe, id, τ, t) est reçu
3:          si il existe un jeton j' = (−signe, id, τ', t') dans Mem alors
4:              Mem ← Mem−j'
5:              AnnulerMinuteur(j')
6:              si signe = + alors Δ = τ' − τ et J = I(p, t') fin si
7:              si signe = − alors Δ = τ − τ' et J = I(p, t) fin si
8:              si Δ ∉ J alors Erreur(temporelle) fin si
9:          sinon
10:             Mem ← Mem+j
11:             DéclencherMinuteur(j,t,p)
12:         fin si
13:     fin Quand

14:     Quand un message timeout(j) est reçu
15:         Mem ← Mem−j
16:         Erreur(jeton mort)
17:     fin Quand
18: fin procédure
```

Algorithme 2: Procédure exécutée par une place évènementielle p

positif, ce jeton négatif finira lui aussi par être considéré comme mort ; au final on obtient deux erreurs de jetons morts au lieu d'une seule erreur temporelle. En aucun cas, une exécution fautive peut passer pour correcte auprès du moniteur et ceci quelque soit la validité de notre hypothèse affirmant l'inexistence de jeton pouvant dépasser la date limite.

Nous allons dans cette partie détailler et justifier les bornes temporelles garantissant un délai maximal d'acheminement d'un jeton et qui donc permettent d'affirmer que le jeton considéré ne sera jamais reçu dans les temps, et, si jamais il était reçu, sera de toute façon responsable d'une erreur temporelle. Pour cela nous allons considérer deux cas. Dans le premier cas, un jeton positif a été reçu et le jeton négatif correspondant est attendu ; dans le second cas, un jeton négatif a été reçu et le jeton positif correspondant est attendu.

Hypothèses temporelles Il est bien évidemment impossible de définir de telles bornes sans hypothèses sur le délai d'acheminement des messages. On supposera que chaque lien est synchrone.

Définition 21 (Délai d'acheminement des messages).

On notera $\delta(t)$ le délai maximum entre l'apparition d'un évènement (t, τ, id) et la réception du message correspondant par la place t ; dit autrement cet évènement sera reçu par t au plus tard à la date $\tau + \delta(t)$. De même un message envoyé à la date τ par une transition t à une place p sera reçu par cette place avant $t + \Delta(t, p)$.

Attente d'un jeton négatif Nous allons dans cette partie considérer la plus longue attente nécessaire en considérant que le jeton négatif est correct. Soient $e_1 = (t_1, \tau_1, id_1)$ et $e_2 = (t_2, \tau_2, id_2)$ les évènements correspondant respectivement au jeton positif et au jeton négatif. Le pire cas correspond au scénario dans lequel les messages correspondant au jeton positif sont reçus instantanément et les messages correspondant au jeton négatif sont reçus après le délai maximum.

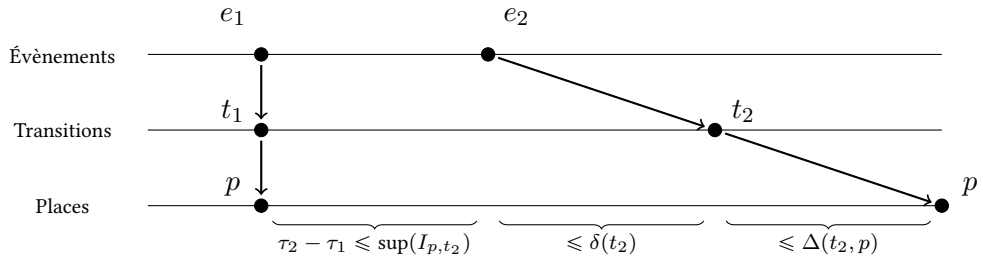


Figure III.10.: Calcul du plus long délai dans le cas de l'attente d'un jeton négatif

Comme on peut le constater dans la figure III.10, dans le pire des cas, l'évènement e_1 sera reçu instantanément par la transition t_1 et le jeton positif correspondant sera reçu instantanément par la place p, c'est-à-dire à la date τ_1.

L'évènement correspondant au jeton négatif se fait dans un intervalle de temps $I(p, t_2)$ après τ ; le temps d'acheminer le message vers t_2 puis vers p, le message correspondant au jeton négatif sera reçu au plus tard à la date $\tau + \sup(I_{p,t_2}) + \delta(t_2) + \Delta(t_2, p)$. Cependant, la place p, ayant simplement reçu le message correspondant au jeton positif, ne sait pas quelle transition va être responsable du jeton négatif ; dit autrement la place p sait par quelle transition le jeton positif a été créé, mais elle ne sait pas encore quelle transition va générer le jeton négatif. Pour cette raison il faut considérer la formule précédente, non plus avec t_2, mais avec une transition t quelconque de $p^\bullet$, puis de prendre le maximum.

$$T_+(p) = \max_{t \in p^\bullet} \{\Delta(t, p) + \delta(t) + \sup(I_{p,t})\} \tag{III.2}$$

Attente d'un jeton positif Nous allons maintenant considérer le cas opposé où le jeton négatif est reçu en premier et où le jeton positif arrive après. Le pire cas correspond aussi au scénario où le premier jeton reçu — ici, le jeton négatif — est reçu instantanément et où le second jeton reçu, l'est après le délai maximal.

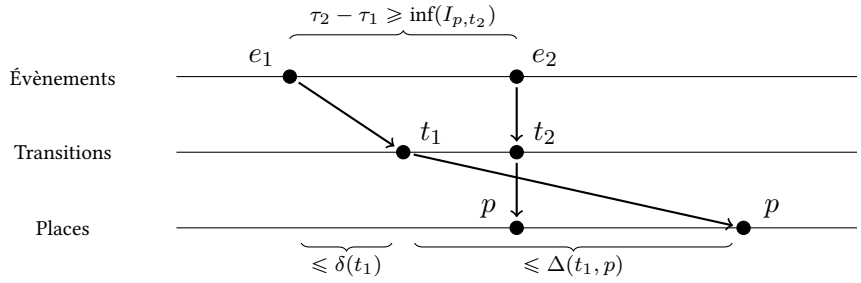


Figure III.11.: Calcul du plus long délai dans le cas de l'attente d'un jeton positif

Comme on peut le constater dans la figure III.11, dans le pire des cas, l'évènement e_1 sera reçu instantanément par la transition t_1 et le jeton positif correspondant sera reçu instantanément par la place p, c'est-à-dire à la date τ_2. Le pire cas correspondant au cas où le délai entre la réception du jeton négatif et celle du jeton positif est maximal, il nous considérer le délai minimal entre les deux évènements. Ainsi, si un jeton négatif est reçu à la date τ_2 sa contrepartie positive sera reçue au plus tard à la date $\tau_2 - \inf(I_{p,t_2}) + \delta(t_1) + \Delta(t_1, p)$. Puisque que l'on ne sait pas de quelle transition t_1 viendra le jeton positif, il nous faut considérer là aussi le maximum pour les transitons t de ${}^\bullet p$.

$$T_-(p, t_2) = \max_{t \in {}^\bullet p} \{\Delta(t, p) + \delta(t)\} - \inf(I_{p,t_2}) \tag{III.3}$$

Algorithme Une fois ces considérations faites, l'algorithme du minuteur consiste simplement à utiliser les formules ainsi calculées. Notre moniteur est créé avec comme argument un jeton et, suivant le signe de ce dernier, on applique la formule correspondante. Une fois l'attente terminée, il suffit d'envoyer un message *timeout* à la place qui a déclenché le minuteur.

```
1: procédure MINUTEUR(jeton,t0,p)
2:     si jeton est positif alors
3:         T = max_{t∈p•} {Δ(t,p) + δ(t) + sup(I(p,t))}
4:     sinon
5:         T = max_{t∈•p} {Δ(t,p) + δ(t)} − inf(I(p,t0))
6:     fin si
7:     attendre T
8:     p ! timeout(token)
9: fin procédure
```

Algorithme 3: Procédure associée au minuteur

Cette algorithme ayant été présenté, le protocole est maintenant entièrement décrit.

3.5. Redondance et tolérance aux fautes

Notre approche est naturellement fortement distribuée. Un unique fil d'exécution est associé à chaque place évènementielle et chaque transition évènementielle. Cependant, il peut être intéressant d'étudier le comportement de notre moniteur dans le cas d'arrêt inopiné d'un nœud et les différentes politiques de redondance nous permettant de rendre notre moniteur davantage tolérant aux fautes.

3.5.1. Transitions évènementielles

Un arrêt inopiné du processus associé à une transition évènementielle correspond à une série d'évènements manqués. Cependant, le principe même de notre moniteur est de gérer au mieux cette situation : une erreur sera relevée suite à la mort d'un jeton et le moniteur continuera à fonctionner.

La redondance ne pose aucun problème ; les transitions évènementielles étant simplement des intermédiaires qui routent les évènements vers les places. Il suffit d'ajouter un mécanisme qui permet d'ignorer un évènement déjà reçu. Un tel mécanisme ne nécessite pas une mémoire infinie dans le cas synchrone que nous considérons dans cette thèse.

En effet pour ne pas tenir compte deux fois d'un même jeton, il suffit pour les places de stocker les jetons reçus dans une autre mémoire où ils ne seront pas supprimés lors de la simplification de deux jetons.

Mais une telle approche peut rapidement faire diverger en mémoire dans le cas où une infinité de jetons serait reçue.

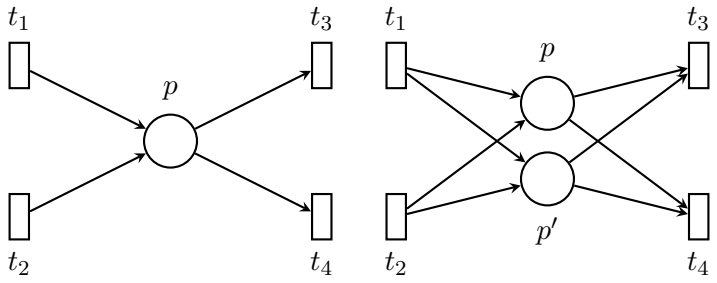
Cependant, puisque nous avons fait l'hypothèse de liens synchrones, à partir du moment où l'on sait qu'un message ne peut pas avoir un retard de plus de Δ, il nous suffit de supprimer les jetons pour lesquels il est impossible de recevoir un *alter ego* négatif dans les temps. Ainsi la mémoire restera bornée à partir du moment où l'on fait l'hypothèse que pendant un intervalle Δ le nombre d'évènements reçus est borné.

3.5.2. Places évènementielles

Les places sont le cœur même de notre moniteur puisqu'elles sont responsables de la détection et du déclenchement des erreurs à partir de leurd observations. Un arrêt inopiné de ces dernières peut être critique dans le sens où des faux positifs (erreurs non détectées) deviennent possibles.

Heureusement, dans ce cas la redondance s'avère assez efficace. En effet chaque place se comportant comme un trou noir (elles ne font que recevoir de l'information et n'en émettent que pour déclencher des alarmes) il nous suffit de les dupliquer. En supposant qu'au moins une copie de la place fonctionne correctement pour interdire tout faux positif. Cela est d'autant plus intéressant que la duplication simple ne nécessite pas de modifier le protocole mais simplement de modifier notre modèle.

Plus formellement, pour dupliquer une place p, il suffit de rajouter autant de place p' que l'on désire vérifiant : $p^\bullet = p'^\bullet$ et ${}^\bullet p = {}^\bullet p'$.



(a) Sans redondance (b) Avec redondance

Figure III.12.: Redondance des places

3.5.3. Redondance optimisée

Si l'on place le nœud qui héberge le processus associé à une place sur un nœud qui contient déjà un capteur et une transition, on aura comme avantage des latences très faibles et des calculs très rapides, mais comme inconvénient d'avoir un système très peu tolérant aux fautes. Une seule erreur matérielle entraîne la mort de nombreux processus de toutes sortes (capteurs, transitions, places, etc.).

Si l'on décide d'héberger un processus par nœud, la tolérance aux fautes sera grandement améliorée, mais au prix d'un coup en latence assez élevé.

Notre approche permet d'avoir le meilleur des deux mondes en ayant des répliques proches des capteurs qui vérifieront rapidement que tout se passe bien et qui seront capable de détecter des retards très rapidement et en ayant d'autres répliques en périphérie du système, moins rapides mais qui permettront de détecter efficacement un problème dans le cœur du système.

3.5.4. Redondance synchronisée

Dans le cas des places évènementielles nous avons fait pour l'instant une redondance assez naïve. L'inconvénient d'une telle approche est que si l'on désire faire de nombreuses duplications de places, la probabilité qu'une duplication ne reçoive pas un jeton augmente avec le nombre de duplicata. Il serait dommage que le redondance ajoute des erreurs plutôt que d'en enlever.

Cependant il est possible de diminuer le nombre de faux négatifs (erreurs détectées à tort) en ne modifiant que la couche de communication et sans toucher aux protocoles ; pour cela il suffit que les transitions diffusent de manière fiable les jetons aux différentes copies d'une même place. Ainsi chaque copie recevra le même ensemble de message et posera donc le même diagnostic.

Pour rappel la diffusion fiable (que le lecteur pourra trouver dans le livre [Ray10] est un protocole simple qui permet de s'assurer que tout les répliques reçoivent le même ensemble de message (pas forcément dans le bonne ordre). Ainsi, si par exemple dans la figure III.12(b) p ne reçoit que le jeton positif et p que le jeton négatif, en utilisant un protocole de diffusion, p et p vont chacun recevoir les deux jetons et aucune erreur ne sera soulevée.

3.6. Résumé

Dans cette partie nous venons donc de voir un premier modèle permettant la supervision d'un système distribué. Pour cela nous avons d'abord étudié la sémantique classique des réseaux de Petri A-temporisés, une généralisation de cette sémantique avec jetons négatifs et enfin un protocole distribué l'implémentant.

Sémantique classique Après avoir rappelé la définition classique des réseaux de Petri A-temporisés avec leur sémantique supposant une observation parfaite, nous avons constaté que cette sémantique n'est suffisante ni pour gérer le cas de messages reçus dans le désordre ni pour gérer le cas de messages perdus. Dans les deux cas, l'exécution du réseau de Petri se trouvait bloquée et le moniteur cessait alors de fonctionner.

Une nouvelle sémantique Nous avons donc proposé l'ajout de jetons négatifs. Ainsi, les évènements peuvent être exécutés dans l'ordre dans lequel ils arrivent et même si certains ne sont jamais reçus. Alors que dans la sémantique classique la validité du franchissement est faite à priori, imposant de recevoir tous les évènements et dans le bon ordre, notre sémantique franchit systématiquement la transition associée à l'évènement reçu et vérifie à postériori que ce franchissement était correct.

Des identifiants Pour être complète, notre approche nécessite l'ajout d'identifiants. Ces identifiants permettent de distinguer deux jetons générés lors de deux occurrences différentes d'un même évènement. Il ne faut pas croire pour autant que les identifiants sont un problème dû à notre sémantique en particulier : ils sont le prix à payer si l'on désire faire de la supervision avec observations imparfaites.

Un algorithme distribué Une des heureuses conséquences de notre approche est qu'elle induit naturellement un protocole fortement distribué. On obtient un processus par place et par transition. Chacun des algorithmes du protocole a été détaillé et la partie la plus délicate à gérer s'avère être la gestion des minuteurs. En effet, plus le délai du minuteur est faible, plus il permet de détecter rapidement des erreurs, cependant un délai trop court peut être la source de nombreuses fausses alertes. Il est donc important que le diagnostic, conformément à la définition 1, ne déclenche pas de fausses alarmes si l'exécution est correcte et si les hypothèses de synchronie sont respectées.

Tolérances aux fautes Si notre protocole est — par construction — tolérant aux fautes du système, il n'est pas forcément tolérant à ses propres fautes. Certes, un arrêt des processus associés aux transitions n'est pas bien grave (une erreur sera quand même déclenchée) mais un arrêt des processus associés aux places peut être catastrophique. C'est pourquoi nous avons détaillé les possibilités de redondance pour tolérer un certain nombre de fautes.

Chapitre IV

Supervision d'un modèle étendu

> Par delà des vagues de toits, j'aperçois une femme mûre, ridée déjà, pauvre, toujours penchée sur quelque chose, et qui ne sort jamais. Avec son visage, avec son vêtement, avec son geste, avec presque rien, j'ai refait l'histoire de cette femme, ou plutôt sa légende, et quelquefois je me la raconte à moi-même en pleurant.
>
> *Le Spleen de Paris — Les Fenêtres*
> Charles Baudelaire

On remarque que dans notre approche, on ne prend jamais l'initiative de tirer une transition ; les transitions à tirer nous étant données via les évènements reçus. L'usage de transition évènementielle est efficace et adapté pour représenter l'état du système. Cependant, on aimerait pouvoir lancer la vérification de certaines propriétés quand un certain état est atteint. Comme nous avons pu le voir dans l'introduction, cela peut se faire en considérant des transitions avec une nouvelle sémantique : les transitions logiques.

Une transition logique n'est pas franchie lorsque l'évènement correspondant est reçu (car elle n'a aucun évènement associé) ; simplement c'est le moniteur qui regarde à chaque changement d'état si cette transition est franchissable, et, le cas échéant, la franchit.

Pour comprendre l'intérêt de cet ajout considérons que l'on désire vérifier la simultanéité de deux évènements : « si t_1 est franchie à l'instant τ_1 et que t_2 est franchie à l'instant τ_2, alors $|\tau_1 - \tau_2| \leqslant \varepsilon$ ». Une telle propriété peut être vérifiée par le réseau de Petri de la figure IV.1. Supposons que t_1 arrive à la date τ et que t_2 arrive à la date $\tau + 2\varepsilon$; dans ce scénario t_2 arrive en retard. La transition logique t_ℓ doit être franchie au plus tard à la date $\tau + \varepsilon$; à partir du moment où cette date est dépassée, le moniteur peut détecter une erreur, ayant un jeton mort dans la p_1.

Le plan de cette partie sera relativement similaire à celui du chapitre III. Dans un premier temps nous redéfinirons notre modèle et sa sémantique lors de la section 4.1 en la généralisant avec les transitions logiques. Dans un second temps, section 4.2, nous définirons de nouveaux protocoles et algorithmes pour tenir compte de ces changements. En pratique les protocoles

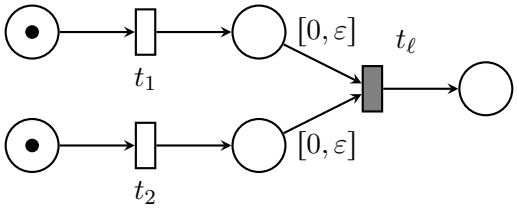


Figure IV.1.: Un réseau de Petri avec transition logique

définis lors de la section 3.4 restent corrects mais ne s'appliquent plus qu'à une sous partie du modèle. Enfin, nous clôturons ce chapitre lors de la section 4.3 par diverses considérations sur la tolérance aux fautes de notre modèle et des différentes politiques de redondance.

4.1. Définitions et Sémantique

Dans cette partie nous allons généraliser la définition 15 concernant les réseaux de Petri de supervision ainsi que leurs sémantiques. Cependant, avant de commencer, il nous faut faire attention que l'ajout d'un nouveau type de sémantique ne crée pas de conflits.

4.1.1. Conflit entre transitions logiques et évènementielles

La principale différence entre les transitions évènementielles et les transitions logiques est que le franchissement des premières est imposé par la réception des évènements quand le franchissement des secondes est décidé par le moniteur. Le fait que deux mécanismes distincts sont susceptibles d'utiliser un même jeton peut être à l'origine de conflits où une transition logique consommera un jeton qui aurait pu être utilisé par une transition évènementielle.

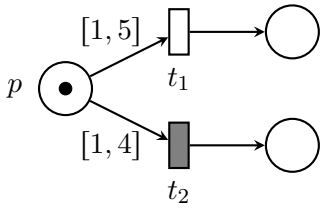


Figure IV.2.: Conflit entre transition logique et évènementielle

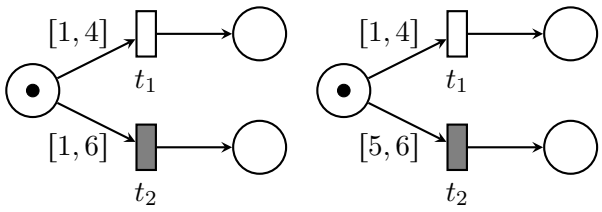
Pour bien comprendre le problème considérons le réseau de Petri de la figure IV.2. La transition t_1 est évènementielle et la transition t_2 logique. On remarque que le moniteur ne peut jamais choisir de tirer t_2 sans risquer de déclencher une erreur ; en effet à aucun moment il ne peut être certain que la transition t_1 ne sera pas tirée par la suite. Prenons un exemple simple. Supposons qu'à la date 1 le moniteur détecte que la transition t_2 est franchissable ; il décide alors de franchir t_2. Cependant, ce même moniteur reçoit un évènement $(t_1, 5)$ peu de temps après. Le jeton ayant disparu de la place p_1, la transition t_1 n'est plus franchissable et une erreur sera soulevée. Si par contre le moniteur n'avait pas franchi t_2 aucune erreur ne serait apparue.

Si l'on résume, le franchissement d'une transition logique peut créer une erreur, alors que sans ce franchissement, l'exécution aurait été parfaitement correcte. Pour éviter de créer des conflits qui n'apporte rien d'un point de vue sémantique, nous allons faire une hypothèse d'absence de conflit entre les deux types de transitions.

4.1.2. Définition

La différence entre la définition dans le cas simple (définition 15) et la définition générale est double. D'abord nous autorisons cette fois un ensemble $T_\ell \neq \varnothing$ et ensuite nous imposons une condition de priorité des transitions évènementielles sur les transitions logiques : une transition logique ne peut pas être tirée si son franchissement affecte un jeton qui pourrait être utilisé par un éventuel évènement correspondant à une transition évènementielle.

Un exemple valant mieux qu'un long discours, nous avons vu que le réseau de Petri de la figure IV.2 ne permettait pas de tirer la transition t_2 de manière sûre. En effet il y a un conflit entre deux transitions, dont l'une – la transition évènementielle – dont on ne contrôle pas le franchissement. La figure IV.3(a) semble davantage correcte ; cependant, on ne peut pas tirer t_2 de manière sûre avant que le jeton ait passé 4 unités de temps dans la place. Pour garantir qu'il n'y ait aucun conflit il est plus sûr que la borne minimale de l'intervalle correspondant à la transition logique soit plus grande que la borne maximale correspondant à la transition évènementielle.



(a) Non valide (b) Valide

Figure IV.3.: Réseau de Petri de supervision

De manière plus formelle, on impose alors la condition suivante : soit t_1 une transition évènementielle, t_2 une transition logique et p une place de ${}^\bullet t_1 \cup {}^\bullet t_2$, alors on a $\sup(I_{p,t_1}) < \inf(I_{p,t_2})$. Dit autrement, une transition logique ne peut pas utiliser un jeton tant qu'une transition évènementielle le pourra encore. En effet I_{p,t_1} représente l'intervalle correspondant à la transition évènementielle ; notons le $I_{p,t_1} = [a, b]$. De même, I_{p,t_2} que nous noterons $[\alpha, \beta]$ représente l'intervalle correspondant à la transition logique. Ce que nous désirons c'est d'avoir $b < \alpha$. Ainsi la transition évènementielle pourra être franchie pendant l'intervalle $[a, b]$ après l'apparition du jeton ; si l'évènement n'a pas eu lieu, alors, à partir de la date α (après l'apparition du jeton), la transition logique pourra être franchie. En ayant deux intervalles bien distincts, ($[a, b] \cap [\alpha, \beta] = \varnothing$), nous savons qu'aucun conflit ne peut avoir lieu et surtout que les évènements sont prioritaires sur le moniteur pour décider d'utiliser un jeton.

Définition 22 (Réseaux de Petri de supervision généralisés).

Un réseau de Petri de supervision généralisé est un tuple (P, T_e, T_ℓ, pre, $post$, p_0, I) où :

- P est l'ensemble des places,
- T_e, l'ensemble des transitions évènementielles,
- T_ℓ, l'ensemble des transitions logiques,
- pre et $post$ deux fonctions de $T = T_e \cup T_\ell$ à valeur dans 2^P
- p_0 représente l'état initial,
- I est une fonction qui à chaque couple $(p, t) \in P \times T$ vérifiant $p \in pre(t)$ associe un intervalle de $\mathbb{R}^+$ et vérifiant pour tout triplet (t_1, t_2, p) de $T_e \times T_\ell \times P$ avec $p \in {}^\bullet t_1 \cup {}^\bullet t_2$, l'inégalité

$$\sup(I(p, t_1)) < \inf(I(p, t_2))$$

4.1.3. Sémantique

Si nous avons déjà défini la notion de *transition franchissable* lors de la description de la sémantique avec transition évènementielle, il nous faut l'adapter au cas des transitions logiques.

Définition 23 (Transition logiquement franchissable).

On dit qu'une transition (t_i, τ_i, id_i) est logiquement franchissable dans un marquage M si, quelque soit $p \in {}^\bullet t_i$, on a un jeton positif d'identifiant $id_i(p)$ apparu a une date τ_p vérifiant :

$$\tau_i - \tau_p \in I(p, t_i)$$

Le lecteur attentif aura remarqué que la définition que l'on vient de donner pour franchissable correspond à la définition 6, à la différence qu'ici, on ne demande plus d'avoir pour tout p de $t^\bullet$, $M(p) = \bot$, c'est-à-dire de ne pas avoir de jeton dans les places suivantes. Le problème est que cette condition, si elle était pertinente, dans le cas des séquences d'évènements, ne l'est plus forcément dans le cas des observations de ces mêmes séquences : dans ce dernier cas un évènement non observé peut laisser croire qu'un jeton est resté dans la place alors qu'il aurait dû être supprimé depuis longtemps. De plus notre ajout des identifiants permet d'avoir plusieurs jetons par place en même temps sans risque de les confondre. La sémantique générale ne faisant pas l'hypothèse des identifiants doit compenser ce manque d'information en imposant un seul jeton par place afin de ne pas les confondre.

La règle de franchissement est assez simple. La première transition franchissable doit être franchie et ceci à partir du moment où elle l'est.

Définition 24 (Transition franchissable prioritaire).

Soit M un marquage associé à une date τ, on définit la transition franchissable prioritaire comme étant le couple (t_i, τ_i) tel que :

- t_i est logiquement franchissable à la date τ_i
- aucune transition logique n'est logiquement franchissable entre les dates τ et τ_i.

Cette définition permet de nous donner une condition pour franchir les transitions logiques. Cependant avant de définir l'ensemble des exécutions valides, et donc en particulier des règles exactes de franchissement des transitions logiques, il nous faut définir la sémantique du franchissement. Fondamentalement, franchir une transition logique a le même effet que franchir une transition évènementielle.

Définition 25 (Simplifié d'un réseau de Petri de supervision).

Soit $\mathcal{P} = (P, T_e, T_\ell, pre, post, p_0, I)$ un réseau de Petri de supervision. On définit le simplifié de $\mathcal{P}$ noté $\mathcal{P}^*$ par :

$$\mathcal{P}^* = (P, T_e \cup T_\ell, \varnothing, pre, post, p_0, I)$$

Dit plus simplement, le simplifié d'un réseau de Petri de supervision est le même réseau de Petri dans lequel les transitions logiques sont remplacées par des transitions évènementielles.

Une exécution d'un réseau de Petri supervisable $\mathcal{E} = e_1 \dots e_n$ est une séquence d'évènements dont certains sont observés (ceux correspondant aux transitions évènementielles) et dont les autres sont générés (ceux correspondant aux transitions évènementielles). On a expliqué dans le chapitre précédent comment calculer l'état obtenu après $\mathcal{E}$ dans le cas particulier où toutes les transitions sont évènementielles ; on peut alors définir l'état obtenu dans le cas où certaines transitions sont logiques.

Définition 26 (Sémantique).

Soit $\mathcal{P}$ un réseau de Petri supervisable et $\mathcal{E} = e_1 \dots e_n$ une observation supervisable. L'état obtenu après l'exécution de cette séquence est l'état que l'on aurait obtenu en exécutant cette séquence sur le simplifié $\mathcal{P}^*$.

Si l'on peut en pratique exécuter n'importe quelle séquence d'évènements correspondant à une observation supervisable, en pratique seul un sous-ensemble peut être généré par notre moniteur. Une suite d'évènements correspondant à des évènements observés et à des évènements générés sera appelée *séquence d'évènements de supervision*.

Avant d'introduire la notion de *séquence d'évènements de supervision*, introduisons certaines notations. Soit $\mathcal{E} = e_1 \dots e_n$ une observation supervisable d'évènements. On note $\mathcal{E}_o$ la suite

extraite des évènements observés et $\mathcal{E}_\ell$ la suite des évènements générés par le franchissement des transitions logiques. Dans $\mathcal{E}_o$ et $\mathcal{E}_\ell$ l'ordre des évènements est conservé.

Définition 27 (Séquence d'évènements de supervision).

Soit $\mathcal{E} = e_1 \dots e_n$ une séquence correct d'évènement. On pose $\mathcal{E}_\ell = f_1, \dots, f_k$.
On dit que $\mathcal{E}$ est une séquence d'évènements de supervision pour un réseau de Petri $\mathcal{P}$ si :

- $\mathcal{E}$ est une observation supervisable d'évènements dans le réseau de Petri $\mathcal{P}^*$;
- Pour tout i, en posant $f_i = (t_i, \tau_i, id_i)$, t_i est la transition franchissable prioritaire dans le marquage M obtenue après l'exécution de la séquence $\mathcal{E}_o, f_1, \dots f_{i-1}$.

D'aucun pourrait légitimement se demander pour quelle raison on a choisi de permuter l'ordre des évènements en mettant tous ceux correspondant à des transitions évènementielles devant. La raison est qu'ainsi nous imposons au franchissement d'une transition logique d'être compatible avec toutes les observations futures. L'observation doit toujours être première par rapport aux choix de générations des évènements.

4.2. Algorithme, protocole et distribution avec transitions logiques

Nous allons dans cette partie montrer comment généraliser le protocole précédent pour y introduire les transitions logiques. Dans un premier temps, nous discuterons sur la distribution des processus associés aux transitions logiques ; en effet il n'est pas souhaitable d'associer comme pour les places et les transitions évènementielles un processus pour chaque transition logique. Dans un second temps nous adapterons la procédure associée aux places pour tenir compte des transitions logiques. Enfin, nous dévoilerons la procédure associée aux transitions logiques.

4.2.1. Distribution du moniteur

On aurait tendance, si l'on suivait l'esprit de la partie précédente, à associer à chaque transition logique un unique processus d'exécution. Cependant, si chaque transition évènementielle s'exécutait dans un fil d'exécution unique, en faisant la même chose qu'avec les transitions logiques, il y aurait des collisions possibles. Considérons le réseau de Petri de la figure IV.4. Trois transitions, t_1, t_2 et t_3, peuvent mettre des jetons dans trois places, p_4, p_5 et p_6. Supposons que les trois transitions puissent être franchies instantanément, dans ce cas, t_5 et t_6 seront franchissables au même instant. Cependant, les franchissements de ces transitions sont mutuellement exclusifs : si le moniteur choisit de franchir t_6, alors t_5 ne peut plus l'être, et réciproquement. La conséquence de ce conflit potentiel est que les processus chargés de décider du franchissement de t_6 et t_5 ne peuvent pas être distincts (sauf à échanger de nombreux messages pour mettre en place un algorithme de consensus au prix d'un coût énorme et inutile).

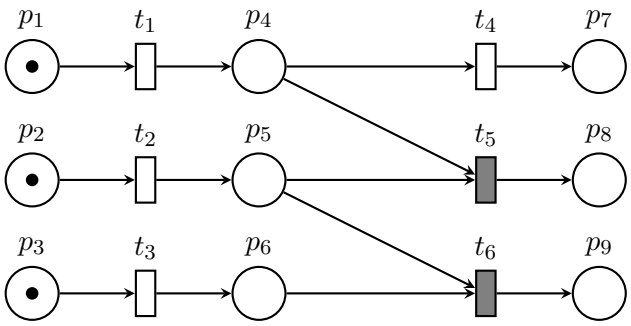


Figure IV.4.: Transitions logiques concurrentes

Définition 28 (Bloc logique).

On appelle bloc logique B_ℓ un ensemble de transitions logiques vérifiant pour toute place p « si $t \in p^\bullet$ alors $p^\bullet \subseteq B_\ell$ »

Le principe du bloc logique est de gérer par un même processus toutes les transitions en concurrence sur un même ensemble de jeton.

Définition 29 (Jetons en conflit).

On dit que deux jetons j_1 et j_2 sont en conflit s'ils appartiennent à deux places p_1 et p_2 précédant un même bloc logique.

Le terme de conflit est utilisé pour exprimer le fait que l'existence de j_1 a une incidence sur la possibilité de franchir j_2. Par exemple si on reprend l'exemple de la figure IV.4, en supposant l'existence d'un jeton dans p_5 et un autre dans p_6, la transition t_6 est alors franchissable. Cependant, l'existence d'un jeton sur la place p_4 peut rendre franchissable la transition t_5, utiliser le jeton de la place p_5, et ainsi rendre le jeton de la place p_6 non franchissable. C'est pourquoi nous disons que les jetons de la place p_4 et ceux de la place p_6 sont bien en conflit.

Définition 30 (Bloc logique minimal).

On dit qu'un blocs logique B_ℓ est minimal s'il n'existe pas de sous-ensemble de E_ℓ strict qui constitue lui-même un bloc logique (dit autrement, E_ℓ ne peut pas s'écrire comme l'union distincte de deux blocs logiques).

Définition 31 (Place évènementielle).

On appelle place évènementielle toute place ne précédant pas un bloc logique. Plus formellement, une place évènementielle est une place p vérifiant :

$$p^\bullet \cap T_\ell = \varnothing \tag{IV.1}$$

Notre moniteur a donc maintenant besoin d'un processus pour chaque transition évènementielle, chaque place évènementielle et pour chaque bloc logique minimal.

4.2.2. Gestion partagée des jetons entre places et blocs logiques

Toutes les places considérées dans la partie précédente (section 3.4 où seules les transitions évènementielles étaient considérées) sont des places évènementielles. Pour le cas de ces places évènementielles, l'algorithme 2 de la section 3.4.3 n'a pas besoin d'être changé.

Définition 32 (Place logique).

Une place qui n'est pas évènementielle est appelée place logique.

Dans la définition des réseaux de Petri de supervision (définition 15 section 3.3.1) nous avons imposé une priorité des transitions évènementielles sur les transitions logiques.

Le fait pour un jeton de ne pas pouvoir franchir une transition logique avant une transition évènementielle est garanti par la propriété introduite dans la partie 3.3.1, propriété affirmant que « pour tout triplet (t_1, t_2, p) de $T_e \times T_\ell \times P$ avec $p \in {}^\bullet t_1 \cup {}^\bullet t_2$, on a l'inégalité $\sup(I_{p,t_1}) < \inf I(p, t_2)$ », ce qui peut se traduire en langage courant par « Soient t_1 et t_2 des transitions de $p^\bullet$ respectivement évènementielle et logique, t_2 ne peut être franchie que lorsque t_1 ne peut plus l'être ».

Cette propriété sera fondamentale pour la gestion des jetons : elle permet de diviser la vie d'un jeton en deux. Dans un premier temps, lorsque le jeton est franchissable uniquement par des transitions évènementielles, il sera géré par le processus de la place logique contenant ce jeton ; dans un second temps, lorsque le jeton devient franchissable uniquement par des transitions logiques — qui par définition appartiennent au même bloc logique — il devient géré par le processus associé au bloc logique correspondant.

Pour les places logiques, la différence d'avec les places évènementielles est double :

- premièrement, elles doivent prévenir les transitions logiques concernées à chaque fois qu'un jeton positif apparaît dans la place ;
- deuxièmement, ces places associent un minuteur à chaque jeton indiquant, non plus la mort de ce dernier, mais le moment à partir duquel il ne peut plus franchir les transitions évènementielles et doit, en conséquent, être géré par le bloc logique.

```
1:  procédure Place(I,L)
2:      Quand un jeton j = (signe, id, τ, t) est reçu
3:          si il existe un jeton j' = (−signe, id, τ', t') dans Mem alors
4:              Mem ← Mem\{j'}
5:              AnnulerMinuteur(j')
6:              si signe = + alors Δ = τ' − τ et J = I(p, t') fin si
7:              si signe = − alors Δ = τ − τ' et J = I(p, t) fin si
8:              si Δ ∉ J alors Erreur(temporelle) fin si
9:          sinon
10:             Mem ← Mem ∪ {j}
11:             DéclencherMinuteurÉvènement(j,t,p)
12:         fin si
13:     fin Quand

14:     Quand un message timeout_event(j) est reçu
15:         Mem ← Mem − {j}
16:         si j est un jeton positif alors logique(j) ! L fin si
17:         si j est un jeton négatif alors Erreur(Évènement) fin si
18:     fin Quand
19: fin procédure
```

Algorithme 4: Procédure exécutée par une place logique p

4.2.3. Procédure associée aux places logiques

Comme nous venons de le voir, l'objectif du minuteur est de détecter l'impossibilité future de tirer des transitions évènementielles, et donc la possibilité de tirer des transitions logiques. Comme le délai est le même que précédemment, il est intéressant de se poser la même question qu'alors : que se passe-t-il si ce délai est minimisé et que le message arrive malgré tout ? Dans ce cas le jeton étant passé au bloc logique, l'arrivé tardive d'un jeton négatif va se terminer par l'attente infructueuse de ce dernier et de sa mort. En d'autre terme, cela peut rajouter des erreurs — et rendre le diagnostic plus difficile — mais une erreur dans nos hypothèses ne peut pas induire la non détection d'un comportement fautif.

Pour en revenir à l'algorithme de la place logique, si le comportement est sensiblement le même que celui des places évènementielles, la gestion du minuteur diffère. En effet, lorsque ce dernier termine, deux cas peuvent se présenter selon le signe du jeton : ou bien le jeton associé est négatif, ce qui signifie que le jeton est mort ou bien ce jeton est positif, ce qui signifie qu'il peut dorénavant être utilisé pour le franchissement d'une transition logique. Dans ce dernier cas, le délai Δ entre la place et la transition logique t correspondante donnée par $\Delta = I(p, t)$ est envoyé au bloc logique.

4.2.4. Procédure associée aux blocs logiques

Un bloc logique, comme nous avons eu l'occasion de l'expliquer est constitué d'un ensemble de transitions logiques. À chaque nouveau jeton reçu, le bloc commence par lancer plusieurs minuteurs. Le premier a pour valeur le délai avant la mort du jeton (ligne 3) ; les autres, lignes 5 à 9, correspondent au délai d'attente associé à chaque transition logique pouvant franchir le jeton. Le principe est de renvoyer le jeton afin qu'il soit reçu par le bloc à une date où l'on sait que le franchissement sera correct. En particulier il faut être certain que tous les jetons antérieurs ont bien été reçus. L'attente correspondante et sa nécessité seront justifiées et détaillées lors de la section 4.2.6.

Ensuite lors de la réception d'un message `jeton`(j), on regarde si la transition t est franchissable (lignes 10 à 21), et si tel est le cas, la transition est franchie (ligne 22 à 25).

Enfin les lignes 27 à 30 gèrent simplement la mort d'un jeton.

Pour cela la procédure maintient deux variables à chaque fois qu'une transition est étudiée. La première variable $franchissable$ reste à vrai tant que les jetons des places précédentes sont présents et avec le bon temps d'attente et elle passe à faux quand un jeton n'est pas présent ou quand son temps d'attente n'est pas valide. La variable J sert à stocker les jetons qui permettent le franchissement de la transition au fur et à mesure que ces jetons sont étudiés.

Si une transition t s'avère être franchissable, alors les jetons correspondant sont supprimés (ligne 23) et les jetons créés par le franchissement sont envoyés aux places de $t^\bullet$ (ligne 24) ; il n'est pas nécessaire de les envoyer aux places de ${}^\bullet t$, puisque les places associées ont oublié les jetons du moment où plus aucune transition évènementielle associée n'était franchissable.

La création des identifiants associés au nouveau jeton se fait via l'appel d'une fonction Id qui à partir des jetons des places de ${}^\bullet t$ génère les jeton des places de $t^\bullet$. Cette fonction doit être donnée avec le modèle. Dans le chapitre V section 5.4, nous discutons cette problématique.

Enfin, quand un minuteur associé à un jeton s'achève (ligne 27), le jeton correspondant est supprimé et une erreur est relevée (lignes 28 et 29).

4.2.5. Des horloges et des délais

Dans la partie précédente, nous avons caché la gestion du temps via un ensemble de minuteurs. Avant de détailler les procédures associées à ces moniteurs, il nous faut d'abord clarifier les hypothèses temporelles ; ces hypothèses étant pour l'instant cachées dans les procédures associées aux différents minuteurs.

Horloges globales et locales Deux hypothèses distinctes peuvent être faites : on peut considérer que les blocs ont accès à l'horloge globale qui permet de dater les évènements, sinon on fait l'hypothèse que chaque bloc a accès à une horloge locale. Dans ce dernier cas, on demande à ce que le temps local avance à la même vitesse pour chacune des horloges locales : si les dates peuvent différer, les durées sont les mêmes pour toutes les horloges du système.

Il est important de remarquer que la date associée au franchissement logique ne change pas, les dates des franchissements des transitions évènementielles étant générées par les capteurs et les dates des franchissements logique étant la conséquence déterministe des premières ; seules les délais de détection du franchissement sont retardés par l'absence d'horloge globale.

```
1:  procédure Logique(L)
2:      Quand un message logique(j) est reçu
3:          DéclencherMinuteurLogique(j)
4:          (p, τ, id) ← j
5:          Pour chaque place t ∈ p• faire
6:              AttendreJeton(j, t)
7:              B ! jeton(j, t)
8:          fin Pour
9:      fin Quand

10:     Quand un message jeton(j, t) est reçu
11:         (+, p_j, τ, id) ← j
12:         Mem(p_j) ← Mem(p_j) ∪ {j}
13:         franchissable ← vrai
14:         J ← ∅
15:         Pour p ∈ •t faire
16:             s'il existe j = (+, t_+, τ_+, id) ∈ Mem(p) tel que τ − τ_+ ∈ I(p, t) alors
17:                 J ← J ∪ {(p, j)}                    ▷ on choisit le j le plus ancien possible
18:             sinon
19:                 franchissable ← faux
20:             fin si
21:         fin Pour
22:         si franchissable = vrai alors
23:             Pour (p, j) ∈ J faire Mem(p) ← Mem(p) − {j} fin Pour
24:             Pour p ∈ t• faire p ! (+, t, τ, Id(J, p)) fin Pour
25:         fin si
26:     fin Quand

27:     Quand un message Timeout_Logique(j) est reçu
28:         Mem(p) ← Mem(p)\{j}
29:         Erreur(logique)
30:     fin Quand
31: fin procédure
```

Algorithme 5: Procédure exécutée par un bloc logique B

Nous allons maintenant détailler deux minuteurs différents : celui associé aux places, et celui associé aux blocs logiques. Il est important de remarquer que le minuteur des places décrit précédemment ne suffit plus, ce dernier ne considérant que les transitions évènementielles.

Délai Nous avions déjà défini pour chaque transition évènementielle t la valeur $\delta(t)$ définie comme le temps maximal nécessaire au processus associé à t pour recevoir l'information

de l'apparition de l'évènement correspondant. Nous avons ainsi pu calculer — en fonction des valeurs différentes de δ pour les transitions évènementielles — les valeurs $\delta_+(p)$ et $\delta_-(p)$ correspondant au temps nécessaire à partir de la date de l'évènement pour recevoir les jetons associés, positifs et négatifs.

Si l'on considère maintenant la présence de transitions logiques, la valeur δ associée à ces transitions est fonction de la valeur de δ pour les places qui précèdent ces blocs qui elle-même dépend de la valeur de δ pour les transitions les précédents. En effet dans le cas des transitions évènementielles, le plus long chemin de communication était de la forme :

$$\text{capteur} \longrightarrow \text{transition évènementielle} \longrightarrow \text{place}$$

Cependant, en présence de transition logique, comme le montre la figure IV.5, les communications peuvent être de la forme :

$$\text{capteur} \longrightarrow \text{transition} \longrightarrow \text{place} \longrightarrow \text{bloc logique} \longrightarrow \text{place} \longrightarrow \text{bloc logique} \longrightarrow \ldots$$

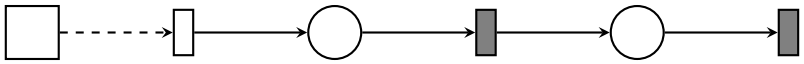


Figure IV.5.: Communication dans un réseau de Petri supervisable

La définition du délai maximal devient ainsi récursive, il est alors malheureusement possible que pour certaines transitions logiques t on ait $\delta(t) = \infty$. En effet il suffit de considérer une boucle comme présenté dans la figure IV.6. À partir du moment où la première transition est franchie, les deux transitions évènementielles peuvent être franchies un nombre infini de fois ; cependant puisque les deux transitions logiques appartiennent à deux blocs logiques distincts, chaque envoi de jeton entre les blocs logiques induit un retard potentiel et la somme de ces retards diverge. Ce problème délicat sera traité par la suite.

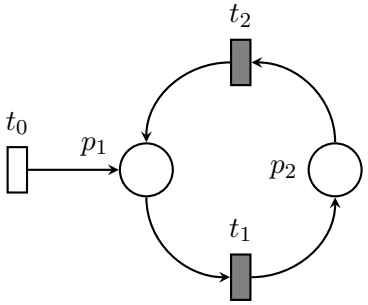


Figure IV.6.: Communication dans un réseau de Petri supervisable

Il nous reste alors deux questions à aborder :

1) Quand faut il prendre acte du franchissement d'une transition ?
2) À partir de quand peut-on décréter la mort d'un jeton ?

4.2.6. Calcul de la date effective du franchissement

Autant les algorithmes 1, 2, 4 et 5 présentés précédemment permettent de calculer la date du franchissement de la transition, autant la date calculée ne correspond pas forcément dans un contexte distribué à la date à laquelle on peut tirer la transition. En effet, une transition peut sembler franchissable à la date τ mais un jeton reçu tardivement peut déclencher le franchissement d'une autre transition rendant la première transition non franchissable. Par exemple dans le schéma suivant de la figure IV.7 la succession des deux évènements $e_1 = (10, t_1, id_1)$ et $e_2 = (15, t_2, id_2)$ impose le franchissement de la transition de t_1 et empêche celui de t_2 ; cependant dans le cas où e_1 est reçu avec retard, la transition t_2 peut sembler franchissable, mais la franchir mènerait à une exécution qui ne correspond pas à la sémantique définie dans la section 3.3. On ne peut donc pas franchir la transition associée à un évènement sans attendre d'être certain d'avoir reçu tous les évènements en conflit potentiel avec une date plus ancienne.

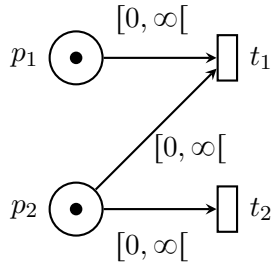


Figure IV.7.: Conflit en amont d'un bloc logique

Au problème de l'attente se rajoute le problème de la synchronisation d'horloge. En effet, il ne faut pas générer un évènement avant qu'il ne se soit effectivement produit. Pour ce faire nous aurons besoin de considérer deux cas, suivant que notre nœud ait accès ou non à une horloge globale.

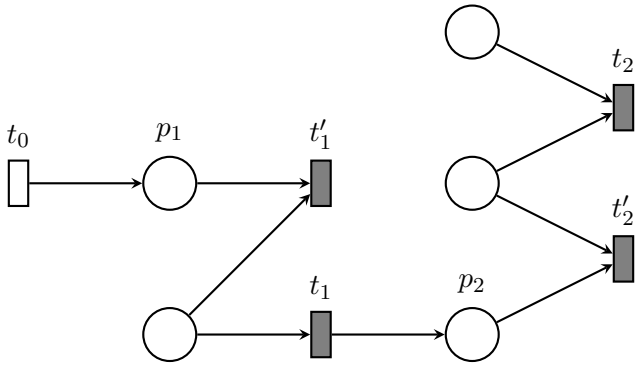
Calcul du délai d'attente maximal

Nous venons de voir que l'on ne peut franchir une transition que si tous les jetons en conflit et plus anciens ont été reçus. Par définition, on dit que deux transitions peuvent avoir un conflit de jeton si elles appartiennent au même bloc logique. Soit B un bloc logique dont t est une transition et p une place de ${}^\bullet t$, la question est alors : en recevant un jeton associé à la date τ sur la place p à partir de quelle date peut-on être sûr d'avoir reçu tous les jetons concernant B associés à une date inférieure à τ ? Pour répondre à cette question il nous faut déjà définir la notion de chemin mixte.

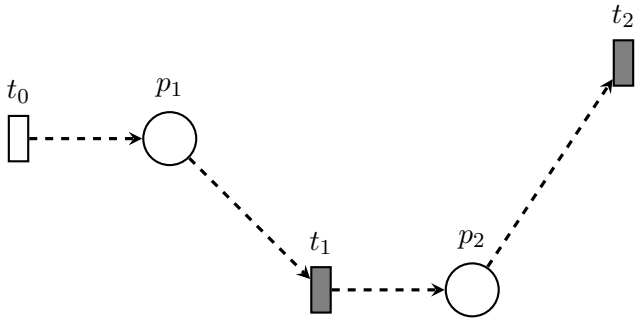
Définition 33 (Chemin mixte).

Une chemin mixte est une suite de places et de transitions de la forme $t_0, p_1, t_1, \ldots, p_n, t_n$ avec :

1) t_0 une transition évènementielle ;
2) $\forall i > 0$, t_i est une transition logique ;
3) $\forall i > 0$, $p_{i+1} \in t_i^\bullet$;
4) $\forall i > 0$, il existe une transition t_i tel que $p_i \in {}^\bullet t_i$ et les deux transitions t_i et t_i appartiennent au même bloc logique.



(a) Le réseau de Petri



(b) Un chemin mixte

Figure IV.8.: Chemins mixtes

Un exemple de chemin mixte est donné figure IV.8. Ce chemin est représenter en vert par t_0, p_1, t_1, p_2, t_2. Ce chemin est bien un chemin mixte car :

1) la première transition est bien une transition évènementielle
2) les transitions suivantes sont des transitions logiques
3) $p_1 \in t_0^\bullet$ et $p_2 \in t_1^\bullet$

4) l'existence d'un jeton dans p_1 peut avoir un incidence sur la possibilité de franchir t_1 (en rendant t_1 prioritaire sur t_1) ; un même rapport existe entre p_2 et t_2.

Un chemin mixte permet de représenter l'origine des retards. Par exemple les franchissements de t_0 ont une incidence sur la date à laquelle p_1 recevra ces jetons, mais cette date aura une influence sur la date à partir de laquelle la transition t_1 pourra être franchie et ainsi de suite. Si le délai avant le franchissement de t_0 est très grand, ce délai devra être pris en considération sur tout le chemin mixte : si t_0 est franchie en retard, alors t_2 le sera aussi.

Définition 34 (Ensemble de chemins mixtes).

L'ensemble des chemins mixtes qui mènent à une transition t est noté $\Gamma(t)$.

Nous pouvons maintenant définir le retard ρ associé à chaque chemin mixte.

Définition 35 (Retard associé à un chemin mixte).

Définissons par induction le retard $\rho(\gamma)$ associé à un chemin γ de Γ.

- si γ est constitué d'une unique transition évènementielle t alors

$$\rho(\gamma) = \delta(t)$$

- si γ est de la forme $\gamma \cdot t \cdot p$ alors

$$\rho(\gamma) = \rho(\gamma \cdot t) + \Delta(t, p)$$

- si γ est de la forme $\gamma \cdot p \cdot t$ alors

$$\rho(\gamma) = \rho(\gamma \cdot p) + \Delta(p, t)$$

Dans cette définition, γ peut correspondre au chemin vide, sans quoi, la récursion ne terminerai jamais.

Au risque de surprendre le lecteur, cette définition est en fait assez intuitive. Le retard associé à une transition évènementielle est simplement le délai maximal nécessaire pour recevoir un évènement, délai que l'on note δ. De plus, le retard associé à un chemin composé, par exemple $\gamma \cdot t \cdot p$, est simplement la somme des retards du début du chemin $\gamma \cdot t$ et du retard associé à la dernière étape du chemin $t \cdot p$.

Si on désire connaître le retard associé à une transition, il nous faut considérer les retards associés aux chemins logiques. En effet dans notre exemple de la figure IV.8, t_1 ne peut pas être franchie tant que tous les jetons de p_1 n'ont pas été reçus, mais t_2 ne peut pas être franchie tant que tous les jetons de p_2 n'ont pas été reçus, eux-mêmes héritant du retard de t_1. Bien

évidemment, nous ne considérons le retard que d'un seul chemin logique, mais puisque plusieurs chemins logiques peuvent mener à une même transition, le retard maximal associé à une transition peut être défini comme suit.

Définition 36 (Attente nécessaire pour le franchissement d'une transition logique).

Le retard maximal associé à chaque transition logique t est donné par :

$$\Delta(t) = \sup_{\gamma \in \Gamma(t)} \{\rho(\gamma)\} \tag{IV.2}$$

On remarque que ces formules peuvent poser problème car rien ne permet d'affirmer que l'ensemble $\Gamma(t)$ est borné (ce qui est le cas par exemple dans la figure IV.6 où il existe une infinité de chemins mixtes de la forme, en langage rationnel, « $t(p_1t_1p_2t_2)^*$ ». Puisque l'on peut obtenir un nombre infini de chemin sur un graphe fini, cela signifie que le retard maximal peut donc lui aussi être infini ! Pour éviter une attente infinie, trois solutions s'offrent alors :

1) soit on interdit la présence de cycle logique (des cycles ne contenant que des places et des transitions logiques) ce qui permet d'avoir des ensembles Γ finis, au prix d'une perte d'expressivité due à une contrainte forte sur la topologie des réseaux de Petri ;
2) soit on ne détecte plus les erreurs évènementielles, aux prix d'un affaiblissement de l'intérêt de notre moniteur, car on ne pourra jamais savoir si la non réception d'un évènement a pour cause l'asynchronie de notre approche ou simplement la perte d'un évènement ;
3) soit — et c'est la solution de loin la plus intéressante — on s'assure que chaque cycle se trouve sur un même nœud.

On considérera dorénavant uniquement ce dernier cas. Dans ce dernier cas, tout chemin de la forme $t \cdot p \cdot \gamma$ sera entièrement situé sur le même nœud à l'exception de la première transition évènementielle. Ainsi le retard associé au chemin $p \cdot \gamma$ noté $\rho(p \cdot \gamma)$ devient nul (car les communications sont instantané au sein d'un même nœud) on obtient alors (et ceci quelque soit le chemin γ) :

$$\rho(t \cdot p \cdot \gamma) = \rho(t \cdot p) + \rho(p \cdot \gamma) = \rho(t \cdot p) = \delta(t) + \Delta(t, p) \tag{IV.3}$$

Ainsi la fonction ρ prend un nombre fini de valeurs sur les ensembles $\Gamma(t)$ où t est une transition logique. Le temps d'attente nécessaire devient alors borné et peut être défini comme suit :

$$\Delta(t) = \max_{\gamma \in \Gamma(t)} \{\rho(\gamma)\} \tag{IV.4}$$

Calcul de la date effective du franchissement d'une transition

Nous allons maintenant déterminer la date à laquelle il faut tirer effectivement la transition. En effet les protocoles, présentés dans les sections 4.2.3 et 4.2.4, nous donnent la date théorique exacte, mais, comme nous venons de le voir, la nécessité d'attendre nous impose de la franchir

réellement plus tard (bien que l'évènement soit estampillé par la date calculée lors du protocole). Le décalage entre franchissement théorique et franchissement réel est plus importante si l'on a à notre disposition uniquement des horloges locales.

Dans les formules suivantes, on considère une transition t franchissable et on note ${}^\bullet t = \{p_1, \ldots, p_k\}$ et $\tau_1 \ldots, \tau_k$ les dates d'apparitions des jetons qui sont placés sur les places de ${}^\bullet t$ et qui rendent la transition franchissable. On se pose alors la question de la date à partir de la laquelle on peut franchir t en toute sûreté.

Horloge globale

Dans le cas des horloges globales, le calcul de cette date est relativement simple. L'algorithme reçoit un jeton et l'on désire savoir combien de temps attendre avant d'avoir le droit de considérer ce jeton. Pour cela il faut être sûr que tous les jetons antérieurs et en conflit aient déjà été reçu.

Le jeton dans la place p_i sera franchissable à partir de la date $\tau_i + \inf(I_{p_i,t_i})$ [1] ; la transition ne sera franchissable qu'à partir du moment où tous les jetons la précédant deviennent tous franchissables ; on obtient en considérant tous les jetons la date donnée par la formule suivante.

$$\max_{1 \leqslant i \leqslant n} \{\tau_i + \inf(I_{p_i,t_i})\} \tag{IV.5}$$

Cependant, comme nous l'avions expliqué précédemment, puisqu'il n'est pas garanti que les jetons soient reçus dans l'ordre, il est possible qu'un de ces jetons doive franchir une autre transition en priorité, une transition dont certains jetons seront reçus en retard. Pour être sûr que ce cas n'arrive pas, il nous faut attendre $\Delta(t)$ comme nous avons pu le voir dans la partie précédente. D'où la formule finale.

Définition 37 (Date de franchissement effective en présence d'horloge globale).

Soit une transition t franchissable par des jetons $j_i, \ldots, j_k$ avec, pour tout i :

1) j_i appartient à une place p_i (qui appartient à ${}^\bullet t$)
2) j_i est apparu à la date τ_i

Alors, avec $\Delta(t)$ le délai défini définition 36, cette transition doit être franchie à la date :

$$\max_{1 \leqslant i \leqslant n} \left\{\tau_i + \inf(I_{p_i,t})\right\} + \Delta(t) \tag{IV.6}$$

On peut facilement réécrire cette date sous la forme d'un maximum :

$$\max_{1 \leqslant i \leqslant n} \left\{\tau_i + \inf(I_{p_i,t}) + \Delta(t)\right\} \tag{IV.7}$$

1. Un jeton sur une place p est dit franchissable par une transition t lorsque son temps d'attente dans la place est inclus dans l'intervalle $I_{p,t}$

L'avantage de cette réécriture est que si chaque jeton $j_i = (p_i, \tau_i, id_i)$ attend la date $\tau_i + \inf(I_{p_i,t}) + \Delta(t)$ avant de se déclarer franchissable, alors la transition ne pourra être franchie que lorsque la date correspondant au maximum sera atteinte. On peut alors décrire le minuteur par l'algorithme 6.

```
1: procédure ATTENDRE_JETON(jeton,t)
2:     Soit (p, τ, id) = jeton
3:     Soit t une transition logique de p•
4:     Attendre la date τ_i + inf(I_{p_i,t}) + Δ(t)
5: fin procédure
```

Algorithme 6: Procédure d'attente de jeton avec l'hypothèse d'une horloge globale

Remarquons que ce minuteur n'attend pas un délai, mais une date. On ne sait pas combien de temps il attendra, mais il arrêtera d'attendre à la date indiquée. Ceci n'est possible que parce que l'accès à une horloge globale nous permet de connaître l'instant indiqué par la date.

À chaque fin de minuteur, notre algorithme ajoute le jeton correspondant. L'ensemble des jeton sera reçu à la date correspondant au maximum des dates des minuteurs, date qui se trouve être exactement celle de la définition 37.

Horloge locale et synchronisation

Une première approche serait d'utiliser les formules précédentes en synchronisant chaque τ_i avec la date locale de la réception du jeton. Par synchroniser, nous entendons associer à chaque date locale une date globale. L'inconvénient de cette méthode est que, les $\Delta(t)$ s'ajoutant, elle est potentiellement divergente en cas de boucle logique ; et ceci quand bien même la boucle est située sur un même nœud.

La solution consiste à considérer la formule suivante :

$$\max_{1 \leqslant i \leqslant n} \{\overline{\tau}_i + \inf(I_{p_i,t_i})\} + \Delta(t) \tag{IV.8}$$

où $\overline{\tau}_i$ est synchronisé

- sur la date de réception du jeton correspondant, si ce jeton a été généré sur un autre nœud
- sur la date de génération théorique du jeton correspondant, si ce jeton a été généré localement.

Par exemple supposons qu'à la date locale 10 on détecte que τ est franchissable à la date globale 20. Alors après attente d'un certain $\Delta(t)$ (par exemple 5), l'évènement est généré en toute sûreté à la date locale 15, cependant on synchronisera la date globale 20 à la date locale 10 (et non 15).

Avant de conclure faisons deux remarques. Premièrement, toutes les formules étant calculées pour marcher dans le pire cas, si on obtient deux synchronisations différentes via deux jetons différents, on considère alors le pire cas ; c'est-à-dire que l'on se synchronise sur l'horloge la

1: **procédure** Attendre_jetons($jeton$,t)
2: Soit $(p, \tau, id) = jeton$
3: Soit t une transition logique de $p^\bullet$
4: Attendre la date $\overline{\tau}_i + \mathrm{Inf}(I_{p_i,t_i}) + \Delta(t)$
5: **fin procédure**

Algorithme 7: Procédure d'attente de jeton avec l'hypothèse d'une horloge locale

plus en retard. Deuxièmement, puisque l'on a imposé que chaque boucle logique est forcément localisée sur un seul nœud, les $\Delta(t)$ ne peuvent pas s'ajouter indéfiniment.

Optimisation

Le calcul de $\Delta(t)$ n'est pas parfaitement optimisé. En effet il considère que chaque jeton apparaissant à une date t sur une place peut être franchi à partir de cette date t. Ce qui n'est pas forcément le cas. On peut utiliser un tel résultat pour diminuer le temps d'attente. Prenons l'exemple de la figure IV.9. Supposons que :

1) le jeton de la place p_2 soit apparu à la date 10
2) le retard maximal d'un jeton sur la place p_1 ne puisse pas excéder 3 unités de temps.

Dans ce cas, il est inutile d'attendre le jeton de la place p_1, car même en supposant que le jeton de p_2 ait été reçu instantanément et que le jeton de p_1 ait un retard maximal, la date la plus tôt associable à p_1 est 7 et donc t_1 ne peut pas être franchi avant la date $7 + 20 = 27$.

Dans tous les cas le jeton de p_2 pourrait être franchi instantanément sans aucun risque.

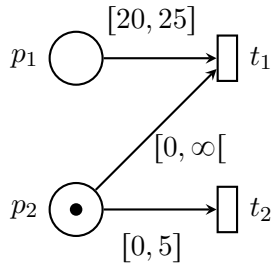


Figure IV.9.: Cas d'une attente inutile

La généralisation de ce principe sur un chemin mixte quelconque nous emmènerait un peu loin et nous nous contenterons des formules précédentes — qui sont, somme toute, correctes à défaut d'être optimales.

4.2.7. Calcul de la mort de jeton

Après cette brève parenthèse revenons à la problématique tragique de la mort des jetons.

Définition 38 (Date de la mort d'un jeton dans un bloc logique).

Un jeton (p, τ, id) meurt une fois qu'il ne peut plus être tiré par aucune transition logique, ce qui nous donne comme date :

$$\max_{t \in p^\bullet \cap T_\ell} \{\tau_i + \sup\{I_{p,t}\} + \Delta(t)\} \tag{IV.9}$$

L'algorithme associé (algorithme 8) ne pose pas de difficulté particulière.

1: **procédure** DéclencherMinuteurLogique(j,Δ)
2: Soit $(p, \tau, id) \leftarrow j$
3: Soit t une transition logique de $p^\bullet$
4: Attendre la date :
$$\max_{t \in p^\bullet \cap T_\ell} \{\tau_i + \sup\{I_{p,t}\} + \Delta(t)\}$$
5: **fin procédure**

Algorithme 8: Procédure d'attente de la mort d'un jeton dans un bloc logique

Cet algorithme a simplement été décri dans le cas d'une horloge globale ; pour obtenir celui qui convient à des horloges locales, il suffit de remplacer « τ_i » par « $\overline{\tau}_i$ ».

4.3. Redondance et tolérance aux fautes

Notre approche est naturellement fortement distribuée. Un unique fil d'exécution est associé à chaque bloc logique, à chaque place évènementielle et chaque transition évènementielle. Ainsi, comme dans la partie précédente, il peut être intéressant d'étudier le comportement de notre moniteur dans le cas d'arrêt inopiné d'un nœud et les différentes politiques de redondance nous permettant de rendre plus tolérant aux fautes notre moniteur.

4.3.1. Places logiques

Si une place évènementielle ne fait que recevoir des jetons, une place logique a aussi pour rôle d'envoyer des jetons au blocs logiques. Le redondance de la partie précédente marche parfaitement mais a « l'inconvénient » pour les blocs logiques de recevoir plusieurs fois un même jeton. Comme pour les places en cas de redondance des transitions, les blocs logiques devront utiliser les identifiants des jetons pour être sûr de ne considérer qu'une fois chaque jeton reçu en plusieurs exemplaires.

4.3.2. Blocs logiques

L'arrêt inopiné d'un bloc logique est aussi grave que l'arrêt d'une place dans le sens où des faux positifs, c'est-à-dire des erreurs non détectées, peuvent facilement apparaître. Une simple

redondance peut par contre être problématique. En effet si deux copies ne reçoivent pas le même ensemble de messages, alors elles peuvent générer des évènements contradictoires voir même considérer comme correcte une exécution fautive.

Une diffusion fiable pourrait faire l'affaire, mais malheureusement, même si toutes les copies reçoivent les mêmes messages, rien ne permet d'affirmer qu'ils seront pris en compte par toutes les copies. En effet, un bloc logique peut déclencher une erreur ne recevant pas un jeton dans les temps, alors qu'une copie aura reçus les deux jetons avant la fin des minuteurs.

Il est alors nécessaire d'utiliser une diffusion fiable et d'algorithme de consensus pour que toutes les copies considèrent le jeton comme arrivé dans les temps ou qu'elles le considèrent comme arrivé tardivement. Cette sur-couche de communication peut sembler très lourde, mais elle est nécessaire pour éviter des incohérences et des erreurs non détectées.

À partir de ce moment là, et pourvu que l'exécution de notre modèle soit déterministe, la sortie sera la même dans tous les blocs logiques.

Cependant, le protocole décrit n'est pas en tant que tel déterministe. En effet, considérons le réseau de Petri suivant. Les transitions t_1 et t_2 sont franchissables exactement au même moment. Pour s'assurer d'une exécution déterministe, il suffit de rendre certaines transitions prioritaire sur d'autres. Pour cela, il suffit de créer un ordre total sur les transitions, en se basant par exemple sur le nom.

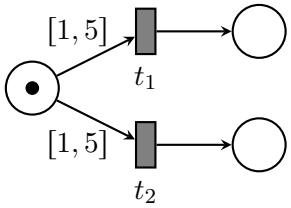


Figure IV.10.: Réseaux de Petri et déterminisme

4.3.3. Redondance optimisées

Faire de la redondance au niveau des blocs logiques aurait comme conséquence un très grand nombre d'instance de consensus et donc des performances limitées. Pour limiter de tels inconvénients, il suffit de faire de la redondance à plus gros grains. Ainsi, si un ensemble de blocs logiques forment un sous-réseau de Petri cohérent hébergé sur un seul nœud [2], alors il suffit de faire des consensus en entrée des copies de ces sous-réseaux de Petri et non plus en entrée de chaque bloc logique.

4.4. Résumé

Ce chapitre présente une généralisation du chapitre précédent dans lequel on ne considérait plus seulement un type de transition (les transitions évènementielles) mais deux (avec les transitions logiques). Nous avons donc suivi le même plan que le chapitre précédent. D'abord la

2. Les exemples du chapitre suivant sont tous de cet forme

définition de la sémantique, ensuite la description du protocole et enfin une discussion sur les politiques de redondance.

Transitions logiques En principe chaque transition correspondait à un évènement. Or, des propriétés relativement simples ne peuvent être facilement exprimables en conservant cette correspondance entre évènements et transitions. L'idée des transitions logiques consiste à permettre à notre moniteur de générer un évènement lorsqu'un certain état est atteint. Contrairement aux transitions évènementielles, les transitions logiques doivent être exécutée dans l'ordre. Un autre point particulièrement important est qu'il faut s'assurer que les transitions logiques et les transitions évènementielles ne rentrent pas en conflit : il ne faut pas franchir une transition logique si un futur évènement peut rendre ce franchissement fautif alors que l'on n'aurait obtenu aucune erreur si le moniteur avait décider d'attendre plutôt que de franchir la transition logique. En séparant clairement pour chaque jeton les moments où le jeton est franchissable par des transitions évènementielles et les moments où il devient franchissable par des transitions logiques, la sémantique s'en trouve être clarifiée et la distribution du protocole simplifiée.

Distribution Comme dans le cas des transitions logiques nous cherchons à découper le protocole en une interaction de multiples processus répartis. Cependant, on ne peut plus seulement avoir un processus par transition. En effet, deux transitions en conflit sur un jeton ne peuvent pas décider en même temps de l'utiliser. Plutôt que de les synchroniser par des protocoles complexes et coûteux en temps nous avons décidé de gérer les transitions en conflit par un unique processus. On parle alors de blocs logiques. Le protocole est distribué avec un processus par place, par transition évènementielle et par bloc logique.

Vie d'un jeton L'algorithme ne pose pas en soit de difficulté particulière si ce n'est la gestion d'un jeton par deux processus distincts. En effet nous avons vu:

1) qu'il était fondamental de ne pas mettre en conflit transitions logiques et transitions évènementielles (concrètement une transition logique ne peut être franchie qu'après avoir dépassé la date à partir de laquelle la transition évènementielle peut être franchie) ;
2) que deux transitions logiques en conflit sont gérées par le même processus en charge du bloc logique correspondant.

Ainsi on peut découper la vie du jeton en deux. Pendant la première partie, pendant laquelle il peut être franchi par une transition évènementielle, le jeton est pris en charge par le processus associé à la place le contenant. Durant la seconde période de sa vie, période pendant laquelle il peut être franchi par des transitions logiques, le jeton est géré par le bloc logique contenant toutes les transitions pouvant l'utiliser.

Gestion des horloges Puisque les évènements associés aux transitions logiques sont gérés par notre moniteur, ce dernier décide donc à partir de quand il doit les franchir. Il est intéressant de constater que notre protocole est sensiblement le même selon que notre moniteur a accès à une horloge globale ou à des horloges locales où le temps avance à la même vitesse pour toutes les horloges.

Redondance Ici encore, la redondance est fondamentale pour s'assurer de la tolérance aux fautes de notre moniteur. Cependant, cette fois-ci la redondance est plus délicate car il est nécessaire que les blocs logiques reçoivent les mêmes jetons en entrée. Il suffit pour cela de mettre en place une communication entre places et blocs logiques, communication basé sur un protocole de diffusion fiable associé à du consensus pour éviter que certains blocs logiques considèrent un jeton comme arrivé trop tard pendant que d'autres blocs l'acceptent.

Chapitre V

Bibliothèque de modèle de propriétés

> Mais hélas ! — car pour le baiser, nos narines et nos yeux sont aussi mal placés que nos lèvres mal faites — tout d'un coup, mes yeux cessèrent de voir, à son tour mon nez s'écrasant ne perçut plus aucune odeur, et sans connaître pour cela davantage le goût du rose désiré, j'appris à ces détestables signes, qu'enfin j'étais en train d'embrasser la joue d'Albertine.
>
> *La recherche du temps perdu — Le Côté de Guermantes*
> Marcel Proust

Nous avons dans les parties précédentes décrit la théorie permettant la mise en place de moniteur temps réel et distribué ; ces moniteurs prenant en entrée des modèles. Si les réseaux de Petri sont un formalisme aujourd'hui bien connu, notre version avec deux sémantiques distinctes pour les transitions reste unique. Ainsi, nous montrerons dans cette partie comment notre formalisme peut être utilisé pour exprimer certaine propriété générale en tirant profit de cette double sémantique.

Dans ce chapitre, nous tenterons d'apporter au lecteur un éclairage sur l'étape de modélisation. Nous tâcherons d'exprimer un ensemble de propriétés, du plus simple au plus complexe, en détaillant le type de transitions nécessaires, le découpage en processus.

Enfin nous terminerons par une discussion sur les méthodes envisageables pour générer des identifiants uniques.

5.1. Des propriétés simples

Dans les exemples suivants, on cherche à exprimer des propriétés en utilisant uniquement des transitions évènementielles. Dans ce cas, on suppose les évènements donnés et l'on complète le réseau de Petri déjà existant par l'ajout de places.

5.1.1. Causalité

Définition 39 (Causalité).

Soient A et B deux évènements, on dit que ces deux évènements sont causalement liés si l'évènement A doit toujours être suivi de l'évènement B.

Dans ce cas il suffit d'ajouter une place, les transitions A et B étant supposées données. Le réseau de Petri de la figure V.1 permet d'exprimer cette propriété. En effet lorsque la transition A sera tirée, un jeton se trouvera sur la place centrale ; ce jeton étant nécessaire pour franchir B on remarque qu'un tel schéma impose à la transition B de ne pas pouvoir être franchie avant la transition A. En revanche, cette propriété étant asynchrone, on ne pose aucune condition sur la date de franchissement de B. En particulier, aucune erreur ne sera soulevée si B n'est jamais franchie. Le cas synchrone fera l'objet de l'exemple suivant.

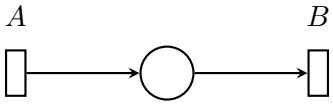


Figure V.1.: Expression de la causalité

5.1.2. Minuteurs

Définition 40 (Minuteur).

Soient A et B deux évènements représentant respectivement le début et la fin d'une action, on désire que l'action s'exécute dans un intervalle donné $\Delta = [0, d]$.

Là encore, cette propriété s'exprime trivialement via un réseau de Petri. Il suffit de rajouter un délai sur l'arête entre la place et le transition B. Cette fois ci, en notant τ la date de transition de A, la transition B ne peut pas être franchie avant la date τ ni après la date $\tau + d$

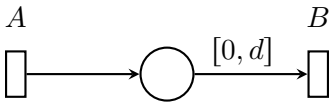


Figure V.2.: Minuteur

5.1.3. Mode normal et mode dégradé

Si les propriétés précédentes restent assez simples, on peut avec des motifs sans transitions évènementielles exprimer des conditions sur des exécutions plus complexes.

Supposons que l'on ait deux modes d'exécutions, le premier correspondant au mode d'exécution normale, le second au mode dégradé. Le principe est de dire que s'il n'est pas possible de garantir que l'exécution normale peut s'exécuter dans un intervalle de temps donné, alors on lance un mode dégradé qui, exécutant moins de chose, peut donc terminer au plus vite.

Définition 41 (Mode normal et mode dégradé).

Soient A et B et C trois évènements, $\Delta_n = [0, T_n]$ et $\Delta_c = [0, T_c]$ tels que :

- A corresponde au commencement de l'exécution d'une tâche ;
- B et C correspondent aux évènements marquant respectivement la fin de l'exécution normale (ne devant pas durer plus de T_n) et de l'exécution dégradée (ne devant pas durer plus de T_c) ;
- $T_c > T_n$, c'est-à-dire que l'on autorise un léger retard dans le cas où le système se met en mode dégradé.

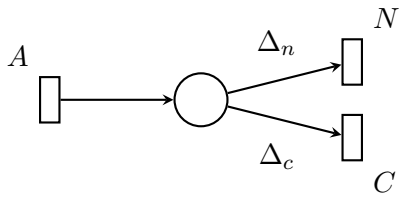


Figure V.3.: Mode normal et mode dégradé

Il faut bien comprendre qu'ici, le moniteur ne décide pas du passage en mode dégradé, mais vérifie simplement que le mode dégradé s'est bien exécuté en cas de retard. Le lecteur ne doit pas s'étonner de voir un mode dégradé plus long que le mode normal ; l'idée étant de déclencher le mode dégradé quand les délais habituels ne peuvent plus être tenus.

5.2. Des propriétés plus complexes

Pour l'instant chacune des propriétés décrites précédemment n'a nécessité à chaque fois que l'ajout d'une place et n'était donc basée que sur des transitions événementielles. Cependant, comme nous avons eu l'occasion de le dire, et comme nous le verrons dans cette partie, l'usage des transitions logiques permet d'exprimer davantage de propriétés.

5.2.1. Simultanéité

La première propriété que l'on cherche à exprimer est celle de la simultanéité de deux ou plus évènements. Évidemment, cette simultanéité ne sera pas exacte, mais à plus ou moins un ε donné.

Définition 42 (Simultanéité).

Soient n évènements $A_1, \ldots, A_n$, alors ces n évènements doivent être simultanés avec un décalage maximum de ε.

Dit autrement, en prenant $n = 3$ si τ_1, τ_2 et τ_3 correspondent aux trois dates associées aux trois évènements, alors on a :

- $|\tau_1 - \tau_2| \leqslant \varepsilon$
- $|\tau_1 - \tau_3| \leqslant \varepsilon$
- $|\tau_2 - \tau_3| \leqslant \varepsilon$

On peut vérifier cette propriété par le réseau de Petri de la figure V.4 en posant :

$$\Delta = \left[0, \frac{\varepsilon}{2}\right]$$

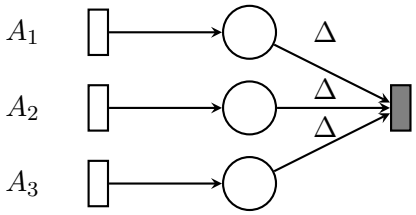


Figure V.4.: Synchronisation temporelle

On remarque que pour vérifier cette propriété, il nous est nécessaire d'introduire une transition logique. On peut bien évidemment généraliser ce motif en synchronisant autant de transitions que l'on désire. Dans tous les cas, ce motif ne contient qu'un bloc logique, lui-même constitué d'une unique transition.

5.2.2. Compteur simple

Comme nous le verrons plus en détail dans la partie suivante, il est intéressant de pouvoir vérifier des propriétés de la forme « *Sur les trois processus, au moins deux ont atteint cette étape* » ; en particulier si on veut faire des systèmes tolérants aux fautes, il est nécessaire d'être capable de formuler l'idée que tous les processus n'ont pas à faire telle action, mais seulement un certain nombre d'entre eux (le nombre minimal de processus corrects autorisés).

L'exemple de la figure V.5 implémente un tel compteur. Plus exactement, à chaque fois qu'une transition A_i est tirée — et on supposent que chacune ne peut être tirée qu'une fois — le compteur s'incrémente. La position initiale du jeton indique qu'au départ aucune transition A_i n'a été tirée. Lorsqu'une de ces transitions est tirées, un jeton apparaît sur la place centrale, et ce jeton doit être tiré immédiatement. On remarque que quelque soit la position du jeton à gauche (dans la place p_0, p_1 ou p_2) la seule transition tirable est celle qui fait disparaître le jeton d'une

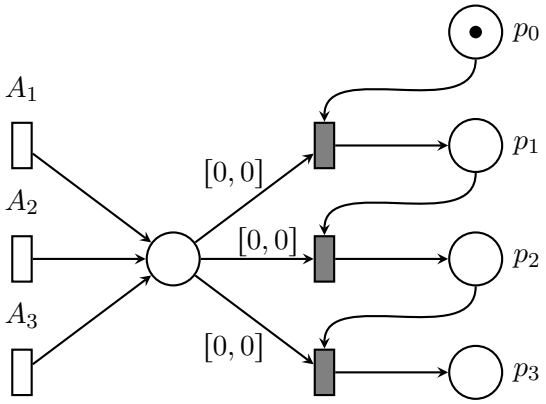


Figure V.5.: Un exemple de compteur

place p_i et qui fait apparaître un jeton dans la place p_{i+1}, incrémentant ainsi le compteur. Dans la configuration finale, si un nombre j de processus ont tirée leur transition A_i, alors le jeton se trouvera dans la place p_j, avec $j \in \{0, 1, 2, 3\}$.

Comme dans l'exemple précédent, ce motif introduit un unique bloc logique qui cette fois est constitué des trois transitions logiques.

5.2.3. Ramasse-miettes

Un des problèmes du réseau de Petri du compteur simple est qu'il marche seulement s'il a été précédemment initialisé. En effet après exécution, des jetons peuvent rester dans les places p_1, p_2 ou p_3 et en particulier il n'y aura plus de jeton dans la place p_0.

Le principe d'un ramasse-miettes est de supprimer, s'il existe, un jeton d'une place et de ne rien faire sinon. Ainsi quelle que soit l'exécution passée, quel que soit l'état de la place avant, à partir du moment où le ramasse-miettes est activé, la place concernée n'aura pas de jeton.

Ce mécanisme est activé via la présence d'un jeton dans une place. Dans le schéma suivant, la place dont il faut supprimer le jeton est la place p, le ramasse-miettes est quant à lui activé dans le cas de la présence d'un jeton dans la place q.

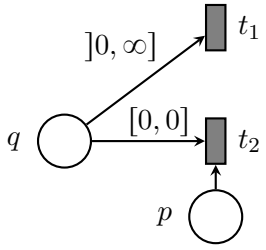


Figure V.6.: Un ramasse-miettes simple

On peut remarquer qu'avec un tel schéma, le jeton apparaissant dans la place q est immédiatement supprimé, que ce soit :

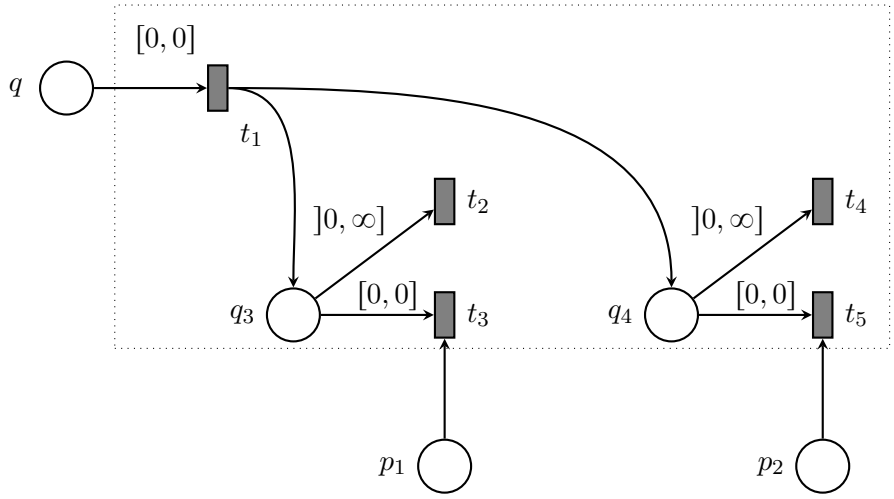


Figure V.7.: Un 2-ramasse-miettes

1) en tirant la transition t_2 et en supprimant le jeton de la p ;
2) en tirant t_1 si la place p est vide.

Pour des raisons de lisibilité et pour éviter d'avoir de trop nombreux ramasse-miettes dans le réseau de Petri final, on peut généraliser ce schéma à n places. Par exemple, la figure V.7 nous montre un 2-ramasse-miettes. Quel que soit le marquage des places p_1 et p_2, l'apparition d'un jeton dans la place q supprime tous les jetons éventuellement présents en p_1 et p_2. Ce schéma se généralise sans difficulté aucune pour n'importe quelle valeur de n. On obtient alors $n + 1$ bloc logique. Dans l'exemple de la figure V.7 représentant un 2-ramasse-miettes on a bien trois bloc logiques : $\{t_1\}$, $\{t_2, t_3\}$ et $\{t_4, t_5\}$.

Définition 43 (Ramasse-miettes).

Un n-ramasse-miettes est un mécanisme qui s'active par l'ajout d'un jeton dans une place et qui supprime dans un ensemble de n places les jetons éventuellement présents.

Pour alléger les schémas, on peut représenter un tel motif par un simple bloc comme représenté figure V.8.

5.2.4. Franchissement atomique

Lorsque l'on cherche à utiliser le compteur simple pour implémenter la propriété « *sur les trois processus, au moins deux ont atteint cette étape* », plusieurs problèmes se posent. Dans le compteur de la figure V.5, on fait l'hypothèse qu'au commencement le jeton est sur la place p_0, puis qu'à la fin de l'exécution — si tous les processus sont corrects — le jeton se trouve

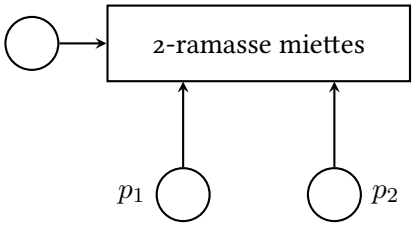


Figure V.8.: Bloc d'un 2-ramasse-miettes

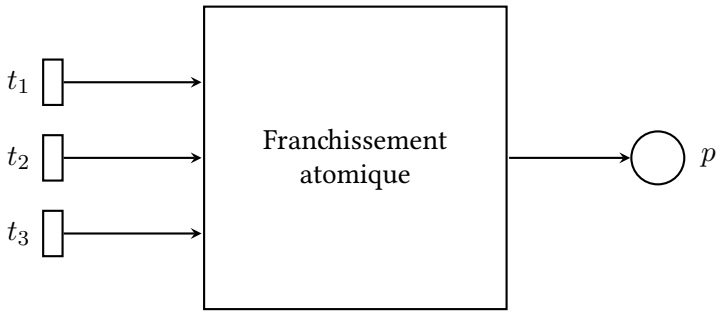


Figure V.9.: Exemple de franchissement atomique (n=3)

sur la place p_3 ; l'inconvénient majeur de cette hypothèse est que le compteur ne peut servir qu'une seule fois. Nous proposons donc dans la figure V.10 un exemple complet dans lequel le compteur est utilisé et remis à zéro régulièrement.

Avant de commencer, décrivons formellement ce que l'on souhaite obtenir. On se donne un ensemble de transitions $t_1, \ldots, t_n$ pour lesquelles on désire qu'au moins k transitions sur ces n soit franchies. Si tel est le cas alors un jeton doit apparaître dans la place p. L'idée est de rendre le franchissement atomique : soit toutes les transitions en sortie sont franchies (et dans ce cas un jeton apparaît dans la place p), soit aucune ne l'est. Un tel mécanisme permet de cacher les erreurs tant qu'elles sont en nombre limité. C'est une façon efficace de gérer la redondance au niveau des capteurs : on ne veut pas déclencher une erreur à chaque fois qu'un capteur a un problème momentané de mesure, du moment où les autres ont un comportement correct. Une erreur globale est déclenchée seulement à partir d'un certain seuil d'erreurs locales. Le lecteur pourra trouver le schéma du bloc correspondant figure V.9.

Définition 44 ((k, n)-Franchissement atomique).

Soient $t_1, \ldots, t_n$ des transitions évènementielles. On souhaite vérifier la propriété suivante : un jeton doit apparaître dans la place p à l'instant exact où k transitions parmi $t_1, \ldots, t_n$ l'ont été. Le compteur des transitions tirées doit, de plus, se remettre à zéro Δ après le franchissement de t_i, où t_i est la première transition franchie.

La seconde propriété concernant la remise à zéro du compteur vient simplement du problème

dont nous parlions en début de cette sous-section. L'exécution d'un réseau de Petri étant cyclique, il est nécessaire d'avoir un mécanisme qui « recharge » le compteur. Le détail du réseau de Petri qui permet d'implémenter la propriété de la définition 44 est donné ci-dessous.

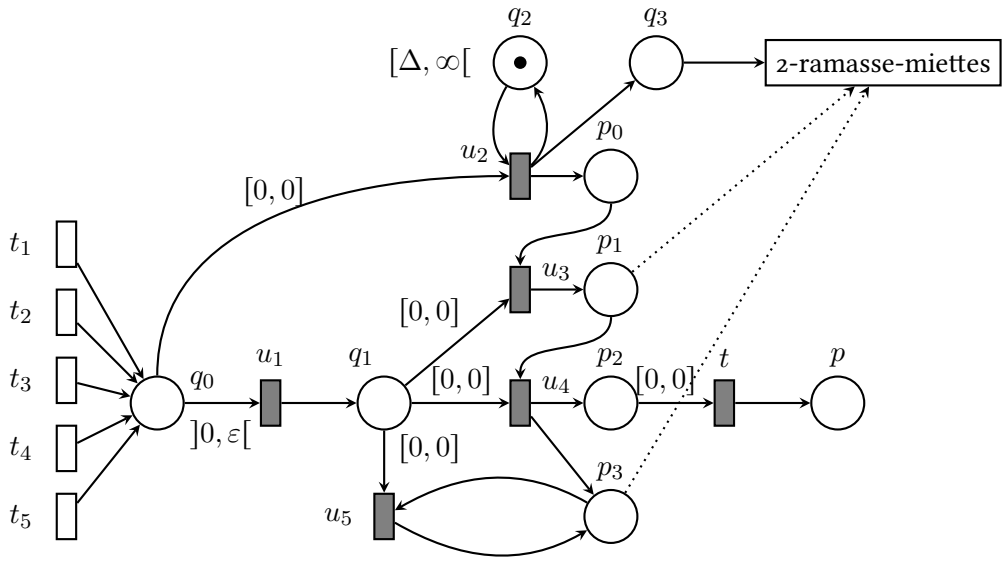


Figure V.10.: Bloc du (3,5)-franchissement atomique

Expliquons plus en détail la figure V.10. Lorsqu'une transition t_i est franchie, le jeton ainsi tiré se retrouve dans une place q_0 de laquelle sort deux arcs.

Le premier arc, étiqueté par $[0, 0]$ permet l'initialisation du compteur. Le franchissement de la transition correspondante permet :

- d'initialiser le compteur en mettant un jeton sur la place p_0 ;
- de mettre en place un « minuteur » qui empêche la transition d'initialisation d'être franchie de nouveau avant Δ_T ; cela veut dire que l'on comptera les jetons qui arriveront dans un intervalle de temps $[0, \Delta_T[$ après l'arrivée du premier.
- de supprimer les jetons restés éventuellement dans les places p_1 et p_3. Par construction, il ne peut pas y avoir de jeton dans les places p_0 et p_2 ; en effet un jeton apparaissant dans ces places franchit automatiquement une autre transition.

Le second arc, étiqueté $]0, \varepsilon[$, ne peut être tiré, que lorsque le premier arc ne le peut plus ; c'est-à-dire après que le réseau a été « nettoyé » et « initialisé ». Le comportement du reste du système est similaire à celui décrit dans la partie précédente sur le compteur ; la seule différence étant que même si plus de trois jetons arrivent, on garde un jeton dans la place p_3. La notation $]0, \varepsilon[$ peut être vue équivalente à l'intervalle $[0, 0]$. La différence étant que les arcs notés $[0, 0]$ sont prioritaires sur les arcs $]0, \varepsilon[$.

Ainsi, à partir du moment où plus de trois transitions ont été franchies, un jeton apparaît dans la place finale p. La conséquence est que le comportement en sortie est binaire : le jeton apparaît si et seulement si au moins deux transitions initiales ont été franchies.

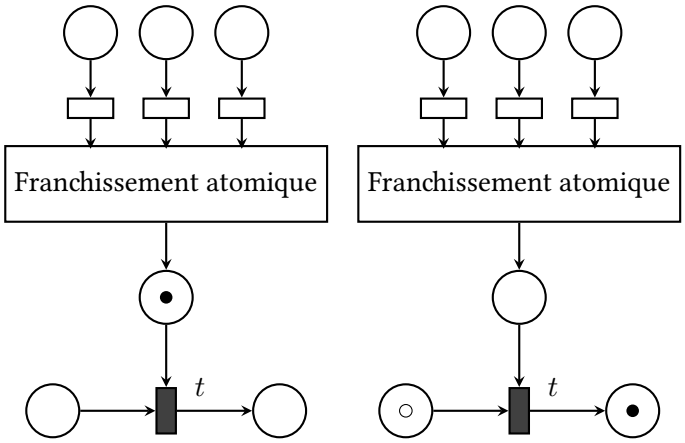
Enfin, il est aisé de voir qu'un tel motif peut être généralisé pour vérifier les propriétés de la forme « *sur les n processus, au moins k ont atteint cette étape* » et ceci quel que soit k et n avec $k \leqslant n$.

On obtient comme bloc logique les quatre ensembles $\{u_1, u_2\}$, $\{u_3, u_4, u_5\}$, $\{u_6\}$, $\{t\}$.

5.3. Transitions génératrices

Dans l'exemple précédent du franchissement atomique, l'évènement « k capteurs parmi n se sont déclenchés » est réprésenté par l'apparition d'un jeton sur une place. Ce choix peux sembler curieux car le franchissement d'une transition logique aurait semblé être plus adapté. Cependant, la sémantique des transitions logiques est différentes des transitions évènementielles ; dans notre cas, la propriété vérifiée correspond à un évènement de haut niveau, évènement qui synthétise les différentes observations en une seule de plus haut niveau.

Pour abstraire plusieurs évènements de bas niveau par un unique évènement de haut niveau, nous introduisons une nouvelle catégorie de transitions. Chacune de ces transitions est déclenchée par l'apparition d'un jeton dans une place et génère des jetons positifs et négatifs, simulant ainsi la réception d'un évènement.



(a) Avant le franchissement (b) Après le franchissement

Figure V.11.: Transitions génératrices

Prenons, figure V.11(a), l'exemple du franchissement atomique. Supposons que deux transitions sur trois ont été franchies, et qu'ainsi, un jeton est apparu en sortie du bloc de franchsissement atomique. Ce jeton va enclencher le franchissement de la transition t selon la sémantique des transitions évènementielles avec ajout de jetons négatifs et positifs. Tous ce passe comme si la transition t avait reçu un évènement. Cet évènement n'ayant pas été reçu, mais ayant été calculé, on parlera de pseudo-évènement.

Ainsi, cette nouvelle catégorie de transitions permet d'abstraire plusieurs évènement en un pseudo-évènement de haut niveau. La sémantique étant extrêmement simple, elle ne rajoute pas de problème théorique : elle n'ajoute ni retard, est peut être gérée par un processus unique.

5.4. Génération des identifiants

Une question importante nécessitant davantage d'explications concerne la génération des identifiants. Il n'y a malheureusement aucun mécanisme général qui permet de générer des identifiants pour tous les réseaux de Petri. Cependant, comme nous avons eu l'occasion de le dire, il ne faut pas voir ceci comme une limitation de notre approche : si de tels identifiants ne peuvent être générés, cela signifie que la propriété ne pourra jamais être vérifiée en présence de fautes ; ce point ayant été discuté dans la section 3.2.4.

Nous allons détailler dans cette partie deux motifs génériques dans lesquels la génération d'identifiant ne pose pas de difficulté majeure. Le premier de ces motifs sera la boucle, le second, les exécutions parallèles de tâches.

5.4.1. En présence de boucle

Considérons le cas d'une transition franchie plusieurs fois du fait de sa présence à l'intérieur d'une boucle, comme le montre l'exemple de la figure V.12. Dans ce cas, il est aisé de générer l'identifiant en prenant le numéro de passage de la boucle. Par exemple, une exécution correcte du réseau de Petri de la figure V.12 — avec comme notation (p, id, s) pour un jeton d'identifiant id, de signe s situé sur une place p — est donné par le tableau 5.4.1.

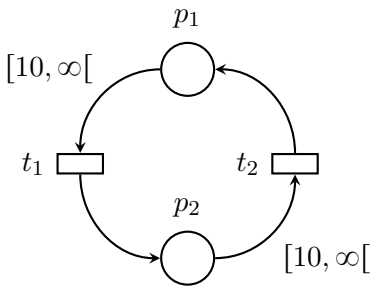


Figure V.12.: Une boucle simple

Un tel système nécessite par contre d'être capable de savoir à chaque instant le numéro de boucle en question. Il est donc nécessaire que la partie du moniteur qui gère la numérotation du tour de boucle soit particulièrement fiable dans sa détection à détecter les débuts et fins de cette boucle ; il peut par contre être moins fiable dans sa communication. Il est moins grave de mal communiquer avec ce capteur, que d'avoir un capteur qui détecte mal les informations. Pour améliorer la fiabilité de la détection des évènements, on peut utiliser de la redondance au niveau des capteurs, redondance rendu transparente par l'usage de notre bloc de franchissement atomique.

5.4.2. En présence de tâches parallèles

Un autre cas classique où une transition peut être franchie plusieurs fois se passe quand la transition peut être franchie par plusieurs fils d'exécution indépendants, chacun correspondant

Exécution	Jetons générés
$start$	$(p_1, 1, +)$
t_1	$(p_1, 1, -)\ (p_2, 1, +)$
t_2	$(p_2, 1, -)\ (p_1, 2, +)$
t_1	$(p_1, 2, -)\ (p_2, 2, +)$
t_2	$(p_2, 2, -)\ (p_1, 3, +)$

Figure V.13.: Identifiants générés

à une tâche avec identifiant. Par exemple, dans le cas d'un serveur, une transition sera franchie pour chaque nouvelle requête. Puisque ces requêtes peuvent être faîtes en parallèle, une bonne pratique pour générer les jetons consistent à utiliser comme identifiant celui associé à la requête.

De même si notre système est composé de plusieurs acteurs (plusieurs robots, serveurs, etc), il suffit d'utiliser l'information de l'identifiant de l'acteur, éventuellement combiné avec le numéro de boucles. En effet, on peut supposer qu'un acteur connaisse son historique (et donc le nombre de passage dans une boucle) mais que la communication soit plus hasardeuse. Prenons un exemple simple, supposons trois robots identifiées par leur adresse IP : 192.168.0.10, 192.168.0.11 et 192.168.0.12. Supposons que ces trois robots accèdent à une boucle. On peut alors utiliser des identifiants de la forme (adresse IP, numéro de boucle). Ainsi pour identifier un jeton concernant la cinquième fois que le second robot utilise le service, on pourra utiliser l'identifiant (192.168.0.11, 5).

5.5. Résumé

Si la théorie sous-jacente de notre moniteur est solide et complète, elle ne suffit pas complètement pour l'utiliser. En particulier, il faut être capable de définir le système et ses spécifications sous la forme d'un réseau de Petri. Cette tâche pouvant être relativement laborieuse, nous avons étudié dans ce chapitre comment simplifier cette étape. Il y a certes une modélisation du comportement du système qui est à faire par les concepteurs, cependant, la partie de la modélisation consistant à spécifier un ensemble de propriétés d'ordre général peut utiliser une bibliothèque de motifs classiques.

Une bibliothèque d'exemples Il serait illusoire de prétendre avoir présenté une bibliothèque exhaustive et complète ; cependant nous avons réussi à montrer à travers un certain nombre d'exemples l'expressivité de nos formalismes. En particulier, il est facile de l'utiliser de manière récursive et modulaire.

Identifiant de jeton La gestion des identifiants peut sembler une partie délicate que nous avons trop rapidement traitée. Si le problème peut sembler difficile dans le cas général, nous avons malgré tout donné une solution au problème des identifiants dans quelques cas classiques. En pratique, si le concepteur ne sait pas quel identifiant utiliser, c'est probablement

parce qu'il n'est pas possible de vérifier les spécifications de son système dans le cadre de nos hypothèses. Sinon, le concepteur peut utiliser l'identifiant qu'il utilise dans son système (une adresse mac, un numéro de requête, etc.). De même dans un cadre clients/serveur, si le serveur peut manquer quelques éléments, chaque client connaît son historique propre et peut aider à générer des identifiants en donnant par exemple le numéro de boucle dans lequel il se trouve.

Chapitre VI

Minotor : un outil complet de supervison

> Drogo explora, fragment par fragment, le triangle visible de désert et il était sur le point de dire qu'il ne distinguait rien de particulier quand, juste au fond, là où chaque image disparaissait dans l'éternel rideau de brume, il lui sembla apercevoir une petite tache noire qui bougeait.
>
> *Le désert des Tartares*
> Dino Buzzati

Nous avons mis en place une implémentation de notre approche afin de tester concrètement sa faisabilité et son efficacité. Notre implémentation, appelée Minotor [1] et écrite en Erlang, reprend les algorithmes présentés précédemment moyennant quelques simplifications en particulier sur les hypothèses temporelles.

Dans un premier temps, dans la section 6.1, nous présenterons nos choix d'implémentation que ce soit sur le langage utilisé, les hypothèses simplificatrices sur les durées de communication, la description concrète de notre modèle ou l'implémentation à proprement parler. Dans un second temps, dans la section 6.5, les questions de l'efficacité et des performances de notre approche seront soulevées.

6.1. Choix du langage

6.1.1. Erlang

Notre approche repose sur un grand nombre de fils d'exécution indépendants, distribués sur un ou plusieurs nœuds, communicant via des échanges de messages. Des langages classiques

1. Pour la petite histoire, le nom de notre outil provient d'une erreur de frappe de l'auteur de cette thèse qui, en anglais, écrivait systématiquement « minotor » à la place de « monitor ». Ayant trouvé l'erreur charmante, c'est naturellement qu'il s'en inspira lorsqu'il dû nommer son outil.

comme le C ou le Java permettent effectivement d'implémenter un tel protocole mais au prix de devoir gérer soi-même les communications entre processus et la gestion des différents fils d'exécution mais surtout, si l'on désire utiliser les *threads* Unix, d'avoir des performances catastrophiques (un *thread* Unix étant relativement lourd). On peut évidemment s'en sortir en implémentant tous les processus d'un même nœud sur un unique *thread* et ré-implémenter le concept de machine virtuelle. Cependant, il existe déjà un tel outil.

L'approche choisie pour Minotor se veut donc pragmatique, dans le sens où elle nous a permis d'implémenter rapidement et efficacement un moniteur avec d'excellentes performances, comme nous le verrons dans la suite. Le langage choisi, Erlang, est en effet un des rares langages qui fournit, au niveau du langage, des mécanismes de haut niveau en ce qui concerne la communication et la distribution des processus. En particulier la communication inter-processus permet de masquer la topologie de notre réseau : deux acteurs communiquent en utilisant le même code, qu'ils soient situés sur un même cœur, sur différents cœurs, ou sur différentes machines.

Erlang est un langage sous licence libre introduit par l'entreprise en télécommunication Ericson dans le but de fournir des systèmes hautement distribué et concurrent, extrêmement tolérant aux fautes , et montant très bien en charge. Avant de présenter notre implémentation il est important de comprendre les propriétés principales du langage et de sa machine virtuelle.

Erlang se base sur la notion d'acteur. Un acteur est *thread* léger, contenant sa propre mémoire et communicant avec les autres via l'échange de mémoire. D'un point de vus applicatif, la communication entre deux acteurs se fait de la même façon que ces deux acteurs soient sur la même machine virtuelle ou sur deux machines distantes d'un même réseau. Enfin les horloges de chaque machine virtuelle sont synchronisées sur celles du système sous-jacent. Ainsi, il faut, par exemple, synchroniser les machines hôtes avec NTP pour que les machines virtuelles Erlang le soient.

6.1.2. Types et syntaxe

Afin de comprendre plus en détail les extraits de code qui seront présents par la suite, il est nécessaire de présenter une très courte introduction à la syntaxe de ce langage.

Types simple Les types de bases en Erlang sont, entre autres, les variables, les chaînes de caractères et les atomes.

- Les variables sont notées avec une majuscule, par exemple `Pong_pid`. En Erlang, chaque variable ne peut être affectée qu'une fois et son contenu n'est donc jamais modifié.
- Les chaînes de caractères entourés de guillemets, par exemple `"fini"`.
- Les atomes. C'est un type propre à Erlang qui représente un mot de manière atomique (et non comme un suite de caractères). Ils sont souvent utilisés dans les échanges de messages. En particulier, dans la suite, ils nous serviront à coder les messages « fini » « ping » et « pong ».

Types complexes Enfin nous avons aussi deux types complexes correspondant :

- aux listes notés, `[a,b,c]`
- aux tuples notés, `{a,b,c}`. Les tuples contrairement aux listes ne peuvent pas être modifiés récursivement par l'ajout ou la suppression d'éléments.

Listes Erlang étant un langage fonctionnel, les listes constituent un type récursif intensément utilisé :

- La notation `[]` correspond à la liste vide et `[Head | Tail]` correspond à la liste dont le premier élément est `Head` et dont le reste est la suite `Tail`. Par exemple, on a l'équivalence : `[1 | [2,3,4,5,6] ] = [1,2,3,4,5,6]`
- l'opérateur `++` correspond à la concaténation de liste et de chaîne de caractères [2]. Par exemple on a :
  1) `"Mino" ++ "tor" = "Minotor"`
  2) `[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]`
- Pour obtenir une chaîne de caractère à partir d'un entier on utilise la fonction `integer_to_list/1` ce qui donne par exemple : `integer_to_list(3) = "3"`.

6.1.3. Présentation pratique d'Erlang via un exemple simple

Pour se donner une meilleure intuition de ce langage, nous allons introduire la syntaxe et les mécanismes d'Erlang via un exemple simple.

Le principe est de faire dialoguer deux acteurs, le premier envoyant des messages `ping` au second qui lui envoie, en retour, des messages `pong` au premier. Chacun des acteurs attendant la réponse de l'autre acteur avant d'envoyer son message. Le premier met fin à l'échange par un message sobre mais efficace : `fini`.

Le code est découpé en module. En général, on associe un module par acteur. Cependant, dans notre exemple, le code associé aux acteurs `ping` et `pong` est tellement simple que l'on n'en utilisera qu'un seul.

Incipit du module Les deux premières lignes servent à indiquer le nom du module et la liste des fonctions publiques ; les nombres suivant le nom d'une fonction correspondent à l'arrité de cette fonction. Par exemple ici, nous avons trois fonctions, la première `start` ne prend aucun argument, la fonction `ping` prendra deux arguments et la fonction `pong` n'en prendra aucun.

Point d'entrée du programme La fonction principale est la fonction `start`. Pour l'exécuter il suffit d'appeler sur un nœud quelconque la fonction `ping_pong:start()`. Cette fonction est assez simple. La première ligne permet de créer un acteur jouant `pong` via la fonction `spawn`. L'identifiant du processus associé à cet acteur sera stocké dans la `Pong_PID` et ce processus exécutera la fonction `pong` du module `ping_pong` sans aucun argument. De même la seconde ligne correspond au lancement du processus associé à `ping` qui prend cette fois-ci

2. En Erlang une chaîne de caractères correspond littéralement à une liste de caractères, d'où la notation commune

```
-module(ping_pong).
-export([start/0, ping/2, pong/0]).

start() ->
    Pong_PID = spawn(ping_pong, pong, []),
    spawn(ping_pong, ping, [3, Pong_PID]),
    ok.



ping(0, Pong_PID) ->
    Pong_PID ! fini,
    io:format("Ping fini~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping a reçus un Pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        fini ->
            io:format("Pong fini~n", []);
        {ping, Ping_PID} ->
            io:format("Pong a reçus un Ping~n", []),
            Ping_PID ! pong,
            pong()
    end.
```

Listing VI.1: Un exemple simple de programme Erlang.

deux arguments : le nombre de message « ping » à envoyer et l'identifiant de l'autre processus. Les identifiants de processus sont des adresses spécifiques à la machine virtuelle Erlang, ces adresses peuvent correspondre à des processus locaux ou à des processus tournant sur une machine virtuelle d'un autre ordinateur, cette distinction étant transparente en ce qui concerne l'usage de ces adresses.

La fonction `ping` Le code de la fonction `ping/2` est répartie en deux blocs, suivant que le premier argument soit nul ou non nul. Le premier argument, appelé N, correspond au nombre de message `ping` qu'il faut encore envoyer.

Dans le premier cas, si N est nul, il suffit d'envoyer le message « `fini` » au processus `pong` puis d'afficher un petit message ; dans le second cas, un couple est envoyé contenant le message « `ping` » et l'adresse du processus Ping que celui-ci peut obtenir grâce à la primitive `self()`. Après cet envoi on attend un message « `pong` », puis lors de sa réception, on affiche un message et la fonction se rappelle récursivement avec une premier argument décrémenté.

Écriture générique des acteurs Après toutes ces considérations, le code de la fonction `pong()` ne pose pas de grande difficulté. En Erlang, dans la majorité des fonctions associées à des acteurs, le code consiste à attendre la réception de messages (`fini` et `ping`) puis en fonction des messages reçus effectuer certains traitements (envoi de messages et affichage d'information) avant de se rappeler récursivement, en modifiant éventuellement ses arguments (la fonction `ping` qui décrémentait la variable N).

6.2. Description du réseau de Petri

6.2.1. Description bas niveau d'un réseau de Petri

Parmi les nombreuses façons de décrire un réseau de Petri, nous avons choisi de faire simple en considérant une liste directement dans le format du langage Erlang. Cette liste contient cinq types différents d'éléments : les places, les transitions, les liens places/transitions d'une part, les liens transitions/places d'autre part et les blocs logiques constitués de transitions logiques. Formellement, on obtient :

- des places représentées par des tuples de la forme

  {`place`, *nom*, *nœuds*}

  où `place` est l'atome `place`, où *nom* est une chaîne de caractère quelconque codant le nom de la place et *nœuds* correspond à l'atome sur lequel devra se trouver le processus associé à cet place.
- des transitions représentées similairement par des tuples de la forme

  {`transition`, *nom*, *nœuds*}

```
petri() ->
    [
     {place, "p1", node3},
     {place, "p2", node3},
     {transition, "t1", node1},
     {transition, "t2", node2},
     {arc, "p1", "t1", 10, infinity},
     {arc, "t1", "p2" },
     {arc, "p2", "t2", 10, infinity},
     {arc, "t1", "p2" },
     {transition, "start", node3},
     {arc, "start", "p_1"}
    ].
```

Listing VI.2: Description du réseau de Petri de la figure III.7, page 32

- des arcs allant des places vers les transitions représentés par de simples tuples

  {`arc`, *nom_place*, *nom_transition* }

  où *nom_place nom_transition* sont les noms d'une place et d'une transition. Les arcs n'étant associé à aucun processus, il n'est pas nécessaire de préciser un nœuds d'exécution.
- des arcs allant des places vers les transitions représentés par des tuples

  {`arc`, *nom_place*, *nom_transition*, *min*, *max* }

  où *nom_place nom_transition* sont les noms d'une place et d'une transition, où *min* est un nombre et *max* est soit un nombre plus grand que *min*, soit l'atome `infinity`.

- Enfin, des blocs logiques représenté un tuple de la forme

  {`bloc`, *liste_logiques*, *nœuds* }

  où liste logique est une liste de transitions logiques de la forme

  {`logique`, *nom_logique* }

Un réseau de Petri étant entièrement défini par ses places, ses transitions et ses arcs, nous pouvons donc le représenter par une liste contenant ces informations. Pour être exact, nous représenterons nos réseaux de Petri par une fonction qui renvoie une liste. L'intérêt de décrire

```
monitor() ->
    [
     {node1,'minotor@site1.laas.fr'},
     {node2,'minotor@site2.laas.fr'},
     {node3,'minotor@site3.laas.fr'}
    ].
```

Listing VI.3: Description de l'emplacement des nœuds

la liste dans une fonction est qu'en pratique il n'est pas nécessaire de décrire le réseau de Petri à la main, mais simplement d'écrire une fonction qui génère pour nous la liste le décrivant. Nous n'imposons aucune restriction sur l'ordre des éléments de la liste, par contre il est fondamental qu'à tout nom indiqué dans les arcs, corresponde une place ou une transition associée.

Le lecteur pourra trouver un exemple dans l'extrait de code VI.2 qui décrit dans une fonction `petri()` le réseau de Petri de la figure III.7. Il est intéressant de remarquer que par rapport au réseau de Petri décrit nous avons rajouté une transition dénotée `start` qui nous permet d'implémenter le marquage initial. En effet pour lancer notre moniteur, il nous suffira de générer un événement associé à cette transition ce qui aura pour effet de mettre en place l'ajout des jetons correspondant au marquage initial. Cette « astuce » est bien évidement généralisable avec n'importe quel réseau de Petri.

D'aucun pourrait justement remarquer que dans la description des places et des transitions est indiqué un nom de nœud, dans notre cas de la forme `node1..3`. Ces nœuds sont les noms de différentes machines du réseau exécutant notre moniteur. La liste de ces nœuds et de leur adresse sur le réseau est décrite dans une autre fonction dénotée `monitor()` décrite dans le Listing VI.3).

6.2.2. Une description hiérarchique de réseaux de Petri

Décrire manuellement un réseau de Petri complet peut être particulièrement fastidieux. Pour cela il peut être intéressant d'utiliser les réseaux de Petri du chapitre précédant sous forme de fonction afin de modulariser la description du réseau de Petri.

Représenter les blocs par des fonctions Un réseau de Petri étant représenté par une liste, il suffit de représenter un bloc par une fonction qui renvoie une liste. Ainsi pour ajouter un bloc il suffit d'appeler la fonction correspondante et de concaténer le résultat avec la liste courante.

Pour prendre un exemple considérons la fonction associée au bloc « 2-ramasse-miettes » présenté dans le *listing* VI.4. La fonction renvoie une liste contenant des blocs logiques, des places et des transitions évènementielles. On remarque qu'elle prend en entrée les trois places de la figure V.7 et qu'elle contient les trois arcs entre les places q, p_1 et p_2 et le reste du réseau. Les trois places ne faisant pas partie du bloc, elles doivent être définies à l'extérieur de la fonction.

Générer automatiquement des places et des transitions Nous allons nous attarder maintenant sur la fonction responsable de la synchronisation. On remarque que lorsque les bloc de-

```
rammasse_miettes(P1,P2,Q,Name,Node) ->
    [
     {place, Name ++ "_p3", Node };
     {place, Name ++ "_p4", Node };
     {bloc,
       [
         {logique, Name ++ "t1"}
       ],Node};
     {bloc,
       [
         {logique, Name ++ "t2"};
         {logique, Name ++ "t3"}
       ],Node}
     {bloc,
       [
         {logique, Name ++ "t4"};
         {logique, Name ++ "t5"}
       ],Node};
     {arc, Name ++ "t1",  Name ++ "p3"};
     {arc, Name ++ "t1",  Name ++ "p4"};

     {arc, Name ++ "p3",  Name ++ "t2", 0, 0};
     {arc, Name ++ "p3",  Name ++ "t3", 0, infinity};
     {arc, Name ++ "p4",  Name ++ "t4", 0, 0};
     {arc, Name ++ "p4",  Name ++ "t5", 0, infinity};

     {arc, Q , Name ++ t1"};
     {arc, Q , Name ++ t3"};
     {arc, Q , Name ++ t4"}
    ].
```

Listing VI.4: Description du ramasse-miettes de la figure V.7, page 76

```
gen_place(Name,Node,I,J) when I>J->
   [];

gen_place(Name,Node,I,J) ->
   Place = {place,Name++integer_to_list(I),Node};
   [  Place | gen_place(Name,Node,I+1,J)].
```

Listing VI.5: Fonction pour générer des descriptions de places

viennent trop gros, il peut être fastidieux de tout rentrer à la main. Pour cela nous avons créé une rapide sous-fonction qui permet de définir en une seule ligne une ensemble de places dont les noms sont de la forme $nom_i, \ldots, nom_j$. On peut alors définir la fonction en deux cas.

1) Si $i > j$ l'appel de `gen_place(Name,I,J)` renvoie la liste vide ;
2) sinon, on renvoie la liste dont le premier élément correspond à la place nom_i et dont le reste de liste correspond à l'appel récursif `gen_place(Name,Node,I+1,J)`.

Concrètement, l'appel de `gen_place("p",'serveur.laas.fr',2,4)` renvoie la liste :

```
[
 {place, "p2", 'serveur.laas.fr'} ,
 {place, "p3", 'serveur.laas.fr'} ,
 {place, "p4", 'serveur.laas.fr'}
]
```

Utilisation de sous-fonctions en tant que sous-blocs Pour montrer comment utiliser un tel bloc au sein d'un autre, nous allons donner la définition d'un deuxième bloc l'utilisant, celui correspondant à la synchronisation logique et situé listing VI.6.

Pour nommer les différentes places et transitions du bloc, il est nécessaire d'avoir des noms uniques ; dans notre implémentation, l'adressage des noms est global (il n'y a qu'une seul transition `t1` par exemple). Cependant, si l'on désire utiliser plusieurs fois le même bloc, on se retrouve devant un conflit sur les noms.

Ainsi chaque fonction définissant un bloc prend un argument supplémentaire : une chaîne de caractère appelé Nom. Ainsi, avec Nom=minos, les transitions `t1`, `t2` et `t3` par exemple seront renommées `minost1`, `minost2` et `minost3`. Le système est rudimentaire et pourrait être géré de manière transparente, mais il permet de façon simple de résoudre le problème de l'unicité des noms.

La première ligne définie simplement une fonction N ; Erlang étant fonctionnel, on a effectivement le droit d'associer une fonction à une variable. Cette fonction permet de simplifier les notation et d'écrire `N("t1")` plutôt que `Nom ++ "t1"` .

Le code de la fonction est une simple concaténation de cinq listes de la forme :

```
gen_place(...) ++
```

```
gen_place(...) ++
gen_place(...) ++
bloc_rm(...) ++
[...]
```

Les trois premières listes permettent de définir les places `p0`...`p3`, `q0`...`q3` et `r1`...`r3`, la quatrième liste correspond au bloc du ramasse-miettes, quand à la dernière liste, elle permet de définir à la main les blocs logiques et les arcs connectant tout ces places et transitions.

6.3. Déploiement du réseau de Petri et hypothèses de synchronie

6.3.1. Déploiement

En pratique un acteur (un *thread* léger dans la dénomination d'Erlang) sera associé à chacun de ses nœuds. Cet acteur aura pour rôle de créer les différents acteurs associés aux places et aux transitions. En particulier il a pour rôle d'indiquer à chaque transition t l'adresse des places de ${}^\bullet t$ et de $t^\bullet$ et à chaque place p l'ensemble des transitions de $p^\bullet$.

Description statique et dynamique Dans une implémentation précédente, il était possible de créer et de supprimer les places et transitions dynamiquement. Cela a un intérêt certain si l'on désire utiliser notre outil en production car cela permet de mettre à jour les propriétés vérifiées sans avoir besoin de redémarrer le superviseur. Cette propriété d'évolutivité bien que nécessaire dans un système réel a été supprimée dans les versions actuelles car cela compliquait inutilement notre code ; dans le cadre de nos expériences, il était plus efficace de décrire le réseau de Petri statiquement et de le déployer en une fois. Cependant il est intéressant de noter la souplesse qu'offre notre approche. Avoir de nombreux acteurs rend plus simple une implémentation dynamique.

Déploiement des places et des transitions En pratique le déploiement se fait de la manière suivante. Tout d'abord, on suppose que les différents acteurs associés aux nœuds ont été précédemment créés. Ce sont ces nœuds qui créent les places et transitions qui leur sont associés. Toutes les adresses des différentes places et transitions sont stockées globalement (chaque place et chaque nœud renseignant après leur création leur adresse à un serveur de *log*). Ce stockage est ensuite utilisé pour contacter les acteurs concernés lors de chaque création d'arc.

Déploiement des arcs Créer un arc `{arc, "p1", "t1", 0, infinity}` consiste à informer la transition `"t1"` de la présence de cet arc et de l'adresse associée à la place `"p1"`. Le serveur de *log* est alors utilisé pour trouver l'adresse de la transition `"t1"` qui sera envoyé l'adresse de `"p1"`. Cela signifie que durant l'exécution du moniteur, la transition connaît déjà l'adresse de chacune des places voisines et elle n'a plus besoin de contacter le serveur de *log* qui n'a aucune utilité en dehors du déploiement (et du débugage). Enfin la transition `"t1"` envoie à la place `"p1"` l'intervalle $[0, +\infty]$. Si l'arc créé avait été de la forme `{arc, "t1", "p2" }`,

```
synchronisation(T1,T2,T3,L1,L2,L3,Nom,Node,Delta) ->
   N = fun(P) -> Nom ++ P end;

  gen_place(N("p"), Node, 0, 3) ++ gen_place(N("q"),Node,0,3) ++
  gen_place(N("r"), Node, 1, 3) ++ bloc_rm("p1","p3","q3",N("rm"),Node)
  ++ [
   {bloc, [ {logique, N("u2")}; # u2 prioritaire sur u1
            {logique, N("u1")};
           ],Node};
   {bloc, [ {logique, N("u3")};
            {logique, N("u4")};
            {logique, N("u5")}
          ],Node};
   {bloc, [ {logique, N("u6")};
          ],Node};

   {arc, T1, N("q0")}; {arc, T2, N("q0")}; {arc, T3, N("q0")};

   {arc, N("q0"),  N("u1"), 0, 0};  {arc, N("q0"),  N("u2"), 0, 0};

   {arc, N("u1"),  N("q1")};
   {arc, N("q1"),  N("u3"), 0, 0};
   {arc, N("q1"),  N("u4"), 0, 0};
   {arc, N("q1"),  N("u5"), 0, 0};

   {arc, N("u2"),  N("q3")};
   {arc, N("u2"),  N("q2")};
   {arc, N("q2"),  N("u2"), Delta, infinity};

   {arc, N("u2"),  N("p0")};
   {arc, N("p0"),  N("u3"), 0, infinity};
   {arc, N("u3"),  N("p1")};
   {arc, N("p1"),  N("u4"), 0, infinity};
   {arc, N("u4"),  N("p2")};
   {arc, N("u4"),  N("p3")};
   {arc, N("p3"),  N("u5"), 0, infinity};
   {arc, N("u5"),  N("p3")};

   {arc, N("p2"),  N("u6"), 0, infinity};
   {arc, N("u5"),  N("r1")};
   {arc, N("u5"),  N("r2")};
   {arc, N("u5"),  N("r3")};
   ] .
```

Listing VI.6: Synchronisation

aucun message n'aurait du être envoyé à "p2" grâce aux simplification faite sur les hypothèses de synchronicité que nous expliquerons dans la section suivante. Pour faire simple, afin de calculer les temps d'attente une place p doit connaître les transitions t qui la précèdent, car la formule de l'attente dépend des valeurs prisent par $\delta(p,t)$. Or, puisque nous avons introduit un majorant de la fonction δ, le temps d'attente de p sera indépendant de l'ensemble ${}^\bullet p$ et ainsi, il n'est plus nécessaire de le connaître.

6.3.2. Hypothèses de synchronie

Nous avons vu que les temps d'attente dans notre protocole dépendent de deux fonctions Δ et δ. La première définit pour chaque couple (t,p) — où t est une transition et p une place — le temps maximal que prend l'acheminement d'un message entre l'envoi par la transition t à la réception par la place p. La seconde fonction δ définit pour chaque transition t le temps maximal nécessaire entre la génération d'un évènement et sa réception par la transition correspondante. Avec N_p et N_t correspondant respectivement au nombre de places et de transitons ces deux fonctions peuvent prendres à elle deux $(N_p+1)\cdot N_t$ valeurs. Les déterminer une par une est faisable en faisant des mesures sur le réseau mais cela peut s'avérer relativement fastidieux.

Pour simplifier les expérimentations nous avons simplement supposé une borne supérieure D à l'ensemble des valeurs prises par l'expression $\Delta(t,p)+\delta(t)$ quand t et p sont respectivement une transition et une place. Cela nous permet d'obtenir à partir des équations III.2 et III.3 de la section 3.4.4 :

$$T_+(p) \leqslant D + \max_{t\in p^\bullet}\{\sup\left(I(p,t)\right)\}$$

$$T_-(p,t_2) \leqslant D - \inf(I(p,t_2))$$

Quand bien même une place ne communique jamais avec des transitions (elle reçoit des messages mais n'en n'envoie aucun) il lui était malgré tout nécessaire de savoir quelles sont les transitions susceptibles de lui envoyer des messages et ceci afin de calculer les valeurs de T_+ et T_-. Avec notre simplification il n'est plus nécessaire de connaître les transitions de ${}^\bullet p$ (avec p la place en question), mais seulement celles de $p^\bullet$.

6.4. Implémentation des acteurs

6.4.1. Acteurs associés aux transitions évènementielles

Le code des acteurs associés aux transitions évènementielles est donné dans le Listing VI.7 ; il consiste en une simple traduction en Erlang de l'algorithme décrit dans les sections 3.4 et 4.2, ce à quoi s'ajoute la gestion de la création des arcs.

Pour rester simple, l'acteur associé à une transition t est implémenté comme un petit serveur via une fonction récursive `loop`. Cette fonction a trois paramètres — qui représentent à eux trois l'état du serveur — le nom de la transition et deux listes ; la première des deux listes `Pre` stocke l'ensemble ${}^\bullet t$; la seconde, `Post`, stocke l'ensemble $t^\bullet$.

```
-module(transition).                                                    1
                                                                        2
loop(Name,Pre,Post) ->                                                  3
    receive                                                             4
        {event,Tags,Time} ->                                            5
            send_token(Pre,minus,Tags,Time,Name),                       6
            send_token(Post,plus,Tags,Time,Name),                       7
            loop(Name,Pre,Post) ;                                       8
        {new_arc,Place,Pid} ->                                          9
            loop(Name,Pre,[{Place,Pid}|Post]);                          10
        {new_arc,Place,Pid,Min,Max}->                                   11
            Pid ! {new_arc,Name,Min,Max},                               12
            loop(Name,[{Place,Pid}|Pre],Post)                           13
    end.                                                                14
```

Listing VI.7: Algorithme associé aux transitions évènementielles

Lors de la création de l'acteur, ces deux listes sont initialisées par deux listes vides. Le serveur attend alors la réception de message, calcule un nouvel état tout en effectuant certaines actions et finalement s'appelle récursivement avec les deux listes éventuellement mises à jours. Cette implémentation des acteurs est représentative de la programmation Erlang qui est fondamentalement récursive.

Deux catégories distinctes de messages peuvent être reçues :

- des messages de type « `event` » qui correspondent à la réception d'évènement ; ce sont les messages de supervision qui sont au cœur de notre approche.
- des messages de type « new_arc » qui correspondent aux déploiements du moniteur.

Avec ce code, on voit qu'il est facilement possible de rajouter des liens durant l'exécution et qu'il n'est pas très difficile de rajouter la possibilité d'en supprimer à l'exécution. Il suffirait pour cela de rajouter le traitement d'un message de type « remove_arc ».

Déploiement Quand une transition t reçoit une demande de création d'un nouvel arc la connectant à une place, il lui suffit de s'appeler récursivement en mettant à jour les paramètres `Pre` et `Post`. Par exemple, ligne 10, l'appel récursif

```
loop(Name, Pre, [{Place,Pid}|Post])
```

correspond à l'ajout de l'arc entre la place de nom `Place` et la transition courante. La liste `Post` est mise à jour par l'ajout d'un couple contenant le nom de la place `Place` et son d'adresse `Pid` (*Process Id*). Dans ce cas, il est nécessaire avant l'appel récursif de mettre à jour la place en lui envoyant l'intervalle $I(p,t)$; la place `Place` étant à l'adresse `Pid`, l'envoi du message se fait ligne 12 via la commande :

```
Pid ! {new_arc,Name,Min,Max}
```

```
1  start(Timeout,Pid,{plus,Tag,Tp}) ->
2     receive
3       {token,{minus,Tag,Tm},{Min,Max}} ->
4           case timer:now_diff(Tm,Tp) of
5              D when D > Max  ->
6                  Pid ! {too_late,Tag,D};
7              D when D < Min ->
8                  Pid ! {too_early,Tag,D};
9              D ->
10                 Pid ! {ok,Tag,D}
11             end
12     after Timeout ->
13             Pid ! {plus_timeout,Tag}
14     end.
```

Listing VI.8: Minuteur des jetons positifs

Supervision Lors de la réception d'un évènement, des jetons doivent être envoyés aux places des listes `Pre` et `Post`. Les jetons négatifs sont envoyés ligne 6 par

```
send_token(Pre,minus,Tags,Time,Name)
```

Les jetons positifs sont envoyés ligne 7 par

```
send_token(Post,plus,Tags,Time,Name)
```

Enfin, la fonction s'appelle récursivement sans avoir besoin de changer son état.

6.4.2. Places et minuteurs

Contrairement à ce que nous venons de faire avec les transitions évènementielles, nous n'allons pas décrire en détail l'implémentation des acteurs associés aux places. Nous allons simplement nous concentrer sur la vérification des contraintes temporelles ; le code associé étant présenté dans les *listing* VI.8 et VI.9.

Le serveur associé aux places est un simple serveur dont le rôle est de recevoir des jetons. À chaque fois qu'un nouveau jeton est reçu, la place lance un minuteur en créant un nouvel acteur.

Plaçons nous dans le cas où le jeton reçu est positif. Dans ce cas, la fonction associé au minuteur est appelé via :

```
start(Timeout,PlaceId,plus,Tag,Tp)
```

Le premier paramètre *Timout* correspond à la valeur $T_+(p)$, le second correspond à l'adresse du père (nécessaire pour lui communiquer une éventuelle erreur) et le dernier correspond au

```
1
2  start(Timeout, Pid, {minus, Tag, Tm}, {Min, Max}) ->
3      receive
4          {token, {plus, Tag, Tp} } ->
5              case timer:now_diff(Tm, Tp) of
6                  D when D > Max  ->
7                      Pid ! {too_late, Tag, D};
8                  D when D < Min ->
9                      Pid ! {too_early, Tag , D};
10                 D -> Pid ! {ok, Tag, D}
11             end
12     after Timeout ->
13             Pid ! {minus_timeout, Tag}
14     end.
```

Listing VI.9: Minuteur des jetons négatifs

jeton à proprement parler qui est codé sous forme de triplet contenant le signe (ici `plus`), l'identifiant et la date de création.

Le minuteur, une fois lancé, attend la réception du jeton négatif correspondant qui devrait arriver dans l'intervalle de temps [`Min`, `Max`].

Suivant la valeur de `Tm` − `Tp`, c'est-à-dire du temps passé par un jeton sur la place (ligne 4), le minuteur informera la place du résultat. La réponse peut-être soit ok (ligne 10), `too_late` (ligne 6) ou `too_early` (ligne 8). Si le jeton négatif n'est pas reçu avant l'expiration du minuteur, alors, la place est informé par un message de la forme `plus_timeout, Tag` (ligne 13).

De même, la gestion des jetons négatifs, décrite dans l'algorithme VI.9, est suffisamment similaire pour ne pas devoir être détaillée.

6.5. Performance et passage à l'échelle

6.5.1. Description de l'expérience

Nous avons cherché à savoir si notre approche basée sur un grand nombre de fils d'exécution était capable de passer à l'échelle. Pour cela, nous avons mis en place une série de tests sur un simple réseau de Petri, à la topologie régulière, présenté figure VI.1. Ce réseau de Petri est constitué de n sous-réseaux de Petri concurrents (correspondant aux n lignes), chacun correspondant à une séquence de n actions. Nous avons généré des réseaux de Petri correspondants à cette topologie avec n à valeur dans l'ensemble $\{2^k \mid k = 0 \dots 10\}$.

Le but de cette expérience étant de calculer l'empreinte mémoire de notre programme ou la charge processeur, nous avons fait tourner notre code sur plusieurs machines virtuelles Erlang tournant sur une seule machine. Plus précisément, il y avait une machine principale servant à stocker les *logs* et à contrôler le moniteur et une autre sur laquelle le moniteur à proprement parler tournait. Le serveur de *logs* n'ayant d'autre but que le débuggage, il nous a semblé

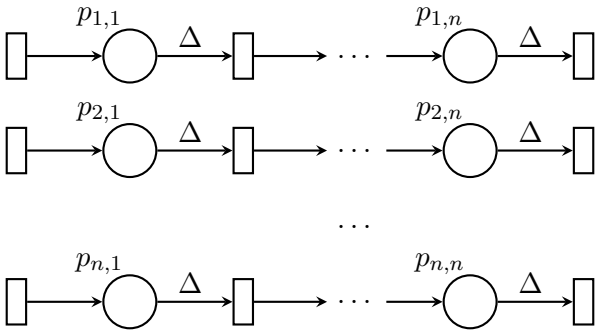


Figure VI.1.: Réseau de Petri de test

pertinent de ne pas en tenir compte dans le calcul de performance.

Le moniteur était hébergé sur un ordinateur HP, possédant 8 cœurs, et faisant tourner Debian GNU/Linux (version Wheezy) de fréquence 2,89 GHz et avec 8 Giga-octet de mémoire. Comme dit précédemment, le moniteur était contrôlé via un second ordinateur, qui se chargeait en plus de générer les évènements.

Remarquons qu'en prenant $k = 10$ l'on obtient un réseau de Petri de 1 048 576 places et de 1 049 600 transitions, c'est-à-dire plus de 2,1 millions de fils d'exécution tournant sur une seule machine. Si notre approche permettait facilement de faire tourner les machines virtuelles sur plusieurs cœurs (il suffit de rajouter une option à la machine virtuelle pour qu'elle se charge de répartir les différents acteurs sur les différents cœurs), pour permettent une plus grande reproductibilité, nous avons fait tourner le moniteur sur un seul cœur afin de faire des mesures sur l'empreinte mémoire.

6.5.2. Empreinte mémoire

Le tableau de la figure VI.2 montre — entre autre — l'empreinte mémoire en fonction de la valeur de n. Plus précisément :

- la première ligne indique la valeur de n ;
- la seconde ligne indique le nombre d'acteurs (en ne comptant que ceux correspondant aux places et aux transitions et qui forme dans notre expérience l'immense majorité) ;
- la troisième ligne indique l'empreinte mémoire en Mega-octet de la machine virtuelle Erlang ;
- la quatrième ligne montre le coût mémoire. En effet, la machine virtuelle Erlang ayant un coût mémoire non négligeable « à vide », il est plus pertinent de considérer la différence entre l'empreinte mémoire obtenue en présence du réseau de Petri et celle « à vide » ;
- enfin, la dernière ligne montre pour des valeurs suffisantes de n un rapport proportionnel d'environ 400 entre le coût mémoire et n.

n	$1 \ldots 32$	64	128	256	512	1024
Nombre d'acteurs	$< 2,1 \cdot 10^3$	$8 \cdot 10^3$	$3 \cdot 10^4$	$1,3 \cdot 10^5$	$5,2 \cdot 10^5$	$2,1 \cdot 10^6$
Empreinte mémoire (Mega-octet)	410	417	435	746	1731	5664
Coût mémoire (Mega-octet)	0	7	25	336	1321	5254
Rapport (coût mémoire) / n	-	-	-	390	397	399

Figure VI.2.: Empreinte mémoire (tableau)

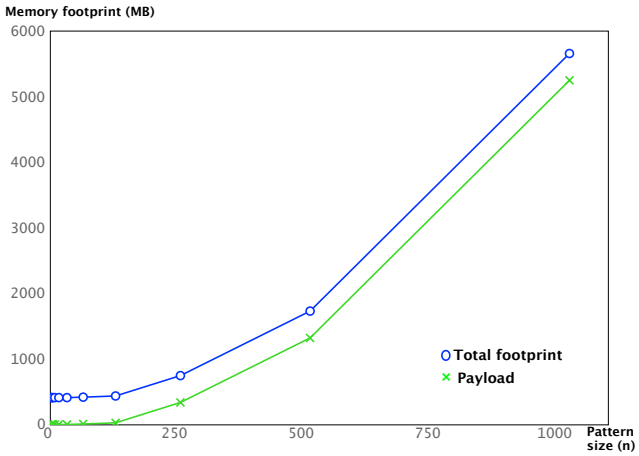


Figure VI.3.: Empreinte mémoire (graphe)

6.5.3. Remarques générales

Pour de petites valeurs de n, l'empreinte mémoire est constante et vaut 410 Mega-octet ; cette valeur correspond à l'empreinte mémoire « à vide » de la machine virtuelle. Cette valeur peut sembler imposante, mais cela est dû à une configuration de la machine virtuelle pour qu'elle soit capable de faire tourner de grands réseaux de Petri (plus de deux millions d'acteurs).

Bien sûr, cette configuration qui correspond à une empreinte mémoire de 410 Mega-octet est beaucoup trop grande pour faire tourner de petits réseaux de Petri ; l'idée était d'utiliser la même configuration pour faire tourner tous les tests. En pratique pour de « petits » réseaux de Petri (avec n=16 c'est-à-dire 496 places ou transitions) une taille mémoire de 30 Mega-octet s'est montrée suffisante.

Nous n'avons pas poussé les expériences avec de plus grandes valeurs de n, car la mémoire à notre disposition (8 Giga-octet) était insuffisante. Le temps de déploiement et de création est conséquent — 40 secondes pour générer les acteurs correspondant aux deux millions de places et de transitions — mais reste correct. De plus nous n'avons pas de chiffre correspondant au temps CPU de l'exécution du moniteur, car les temps de calcul étant quasi instantané par rap-

port au délai de communication engendré par l'envoi et la réception d'évènement, on observe en pratique une activité CPU correspondant à la machine virtuelle quasi nulle.

Enfin si le réseau de Petri était réparti sur plusieurs cœurs et plusieurs ordinateurs, les ressources utilisées seraient la somme des ressources utilisées par les différentes machines virtuelles.

6.6. Résumé

Dans ce chapitre, nous avons étudié et détaillé notre implémentation `Minotor`. Nous avons choisi le langage Erlang pour implémenter notre outil car sa philosophie (dynamisme, grand nombre de processus légers, distribution, etc.) convenait parfaitement pour la création d'un prototype de notre moniteur combinant efficacité et performance.

Description et déploiement du réseau de Petri Nous nous sommes intéressé dans un premier temps à la description du modèle utilisé par notre moniteur. Cette description est fournie par une fonction renvoyant une liste de places, de blocs logiques et de transitions évènementielles. Le principal avantage d'utiliser une fonction est que cela permet de générer automatiquement le réseau de Petri sans devoir rédiger à la main chaque élément. En particulier, on peut tout à fait utiliser l'approche modulaire et récursive décrite lors du chapitre précédent. Notre implémentation se charge alors d'appeler la fonction et de créer tous les acteurs correspondants sur les nœuds spécifiés pour chacun d'entre eux.

Implémentation de l'algorithme et hypothèses de synchronie Pour simplifier l'implémentation nous avons utilisé une borne globale pour l'échange des messages et non plus une fonction donnant le délai maximal de communication pour chaque couple de places et de transitions. L'implémentation retranscrit les algorithmes des parties précédentes dans le langage Erlang.

Performance Les performances de notre implémentation sont très honnêtes et montrent que l'approche consistant à utiliser un grand nombre d'acteurs n'est pas incompatible avec de bonnes performances.

Chapitre VII

Conclusion

Ô cerveaux enfantins !
Pour ne pas oublier la chose capitale,
Nous avons vu partout, et sans l'avoir cherché,
Du haut jusques en bas de l'échelle fatale,
Le spectacle ennuyeux de l'immortel péché

Les Fleurs du Mal – Le voyage
Charles Baudelaire

Nous avons dans cette thèse étudié la problématique de la supervision. Un superviseur (aussi appelé moniteur) est un outil prenant en entrée une observation et qui, à partir de cette observation, relève les éventuelles erreurs. Nous nous sommes concentrés dans cette étude sur la mise en place de tels outils en se fixant trois grands objectifs : 1) La répartition 2) la gestion du temps réel et 3) l'implémentabilité.

7.1. Résumé et résultats

7.1.1. Objectifs

Notre superviseur doit pouvoir être réparti. Cette répartition doit servir à la fois à une plus grande efficacité, mais aussi a une plus grande tolérance aux fautes. Efficacité, parce qu'en répartissant l'intelligence au cœur du réseau distribué, on doit permettre au moniteur de détecter les erreurs au plus tôt. Tolérant aux fautes, car le dysfonctionnement d'une partie de notre moniteur, doit impacter aussi peu que possible le reste du moniteur ; l'idéal étant de rendre ces dysfonctionnements transparents via de la redondance.

Notre superviseur a vocation d'être utilisé dans un contexte temps réel. Nous n'avons jamais imposé d'hypothèses trop fortes sur les délais de communication pour autant. Notre approche de la gestion du temps repose sur deux grands principes.

- Le premier d'entre eux est donné par la définition 1. Concrètement, si les acheminements de messages se font dans les temps, notre superviseur doit poser un diagnostic parfait : il doit détecter une erreur s'il y en a une, et ne pas en détecter s'il n'y a pas. Si par contre les délais ne sont plus respectés (suite, par exemple, à une phase de congestion) on s'autorise à déclencher une erreur, et ceci même si le système s'est comporté de manière correcte. Cela est dû à l'impossibilité de faire la distinction pendant l'exécution entre un message en retard et un message jamais envoyé. Cependant, si le système ne vérifie pas ses spécifications, dans tous les cas, une erreur doit être relevée.
- Le second de ces principes est de détecter les erreurs au plus tôt de manière fiable. Nous ne cherchons pas forcément à décider en respectant une certaine latence (toute erreur doit être détectée au plus tard Δ secondes après son apparition), mais nous avons plutôt pris le problème en sens inverse en donnant les formules permettant de calculer la latence pour chaque propriété vérifiée en fonction de la distribution du superviseur. Ces formules permettent à celui qui met en place le moniteur de le répartir afin d'obtenir la latence souhaitée.

Notre étude doit être implémentable Plus qu'une simple étude théorique permettant de montrer ce qui est possible, nous voulions résoudre tous les problèmes jusqu'à l'implémentation. Ainsi nous avons pu produire une preuve de concept, intitulé `Minotor`, montrant la faisabilité et l'efficacité de notre approche.

7.1.2. Modèles et algorithmes

Comme le lecteur aura pu le constater tout au long de ce manuscrit, on ne peut pas vraiment séparer 1) le modèle représentant le système et ses spécifications, 2) le modèle de fautes et d'observation et 3) le protocole. On choisit en effet un modèle pour ce qu'il permet de faire.

Dans cette thèse, nous nous sommes posés la question de savoir ce qu'il était possible de faire dans le cas où l'observation d'un moniteur était imparfaite. Cela nous a conduit à étudier deux modèles distincts de réseaux de Petri, l'un étant la généralisation de l'autre. Ces modèles nous ont bien permis de remplir tous nos objectifs. En particulier, la version la plus simple offre une distribution particulièrement efficace où chaque élément pouvait tourner dans un fil d'exécution à part ; la version plus riche nécessite par contre de regrouper certains éléments en conflit, ce regroupement pouvant être facilement fait de manière automatique.

7.1.3. Implémentation et propriétés

Une théorie n'étant utile que si elle est pratiquement utilisable, nous avons vérifié de deux manières son utilisabilité réelle. Dans un premier temps en écrivant une preuve de concept en Erlang qui s'est avérée très efficace en pratique ; dans un second temps nous avons étudié la facilité et l'utilisabilité de la description des spécifications du système.

Implémentation L'implémentation a été faite en Erlang, langage dont l'esprit est assez proche de celui de notre protocole ; c'est-à-dire une multitude de processus légers fortement distribués

interagissant entre eux. En particulier, nous avons pu faire facilement tourner des modèles constitués de plusieurs millions de fils d'exécution.

Description Si la description d'un réseau de Petri peut être relativement fastidieuse, il n'en est pas forcément de même pour la description du modèle complet. En effet, il est relativement aisé de le décrire en utilisant une bibliothèque de propriétés. La description se trouve être ainsi modulaire et récursive.

7.2. Futurs travaux et ouverture

Trois pistes de recherche nous semblent particulièrement intéressantes pour continuer et enrichir ces travaux. La première consiste à utiliser un autre modèle d'évènements plus riche que celui considéré jusqu'alors. Le second ajoute d'autres hypothèses d'observation, ce qui aura pour effet de changer du tout au tout la sémantique et les protocoles utilisés. Enfin, la troisième approche consisterait à faire évoluer notre moniteur d'un observateur passif à un chef d'orchestre utilisant ses observations pour commander le système supervisé.

7.2.1. Nouveau modèle d'évènements

Pour l'instant, nous avons utilisé un modèle relativement simple basé sur des évènements ponctuels. Par exemple, dans le cas d'un capteur de température, on pourrait imaginer deux évènements : « la température est supérieure à 50℃ » et « la température est inférieure à 50℃ ». Cependant, d'aucun pourrait être intéressé par des évènements de la forme plus générale « la température est égale à n℃ » où n est un entier naturel. Or, comme nous venons de le voir, la forme générale n'est pas forcément indispensable, car souvent on s'intéresse à des intervalles de valeurs (qui peuvent être représentée par un nombre fini de transitions) plutôt qu'à un continuum. Par exemple ici, nous avions que deux intervalles de température : $]-\infty, 50[$ et $[50, \infty[$. Mais on pourrait aussi imaginer des intervalles $[0, 1[$, $[1, 2[$, $[2, 3[$, etc.

Cependant, une approche contenant trop d'intervalles nécessiterait un nombre d'acteurs trop important pour qu'ils soient tous lancés au démarrage. Grâce au dynamisme de notre approche, on pourrait imaginer que les acteurs concernés ne soient créés que lorsqu'un évènement les nécessite et supprimés lorsqu'ils ne sont plus nécessaires. Ainsi, un réseau de Petri d'un milliard de places et transitions potentielles pourrait au cours d'une exécution normale n'en utiliser qu'une centaine à la fois.

7.2.2. Nouveau modèle d'observation

Tout au long de ce manuscrit, nous avons fait l'hypothèse de la correspondance entre capteurs et transitions évènementielles. Cependant, on peut légitimement considérer cette hypothèse comme étant un peu forte. En effet, dans l'exemple de la section 2.2 rappelé figure VII.1, comment un capteur pourrait différencier la transitiona "AB" (un robot va de la place A à la place B) de la transition "BA" (un robot va de la place B à la place A). Si le capteur et placé sur la porte, les deux évènements sont indiscernables.

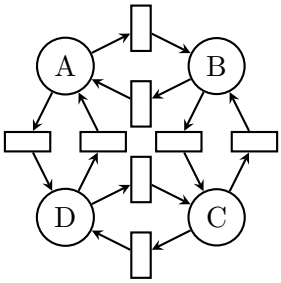
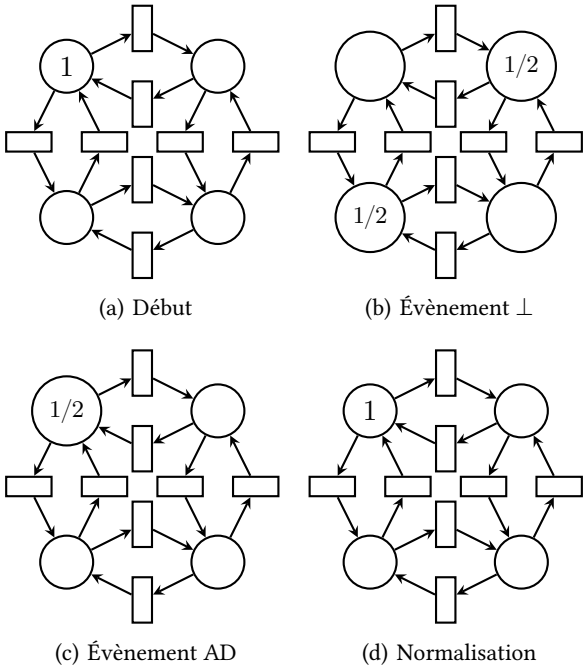


Figure VII.1.: Le réseau de Petri associé



(a) Début (b) Évènement $\perp$

(c) Évènement AD (d) Normalisation

Figure VII.2.: Transitions relatives

Si, comme dans les parties précédentes, on a une correspondance bijective entre évènement et transition, alors les transitions sont dites absolues. Si par contre la donnée de l'évènement ne permet pas de savoir quelle transition évènementielle doit être franchie sans connaître l'état du système on parlera de transitions relatives.

Pour bien comprendre ce qu'un tel changement a comme effet sur la sémantique, il faut prendre conscience qu'une transition absolue nous donne systématiquement l'état final : l'évènement correspondant à la transition "AB" m'indique que le robot était sur la place A et qu'il a rejoint la place B. En fait, on n'a pas besoin de savoir avant la réception de l'évènement où le robot se situe ; cette information nous étant donnée par la transition à franchir. Dans le cas de transition relative, les transitions AB et BA sont associées au même évènement : celui généré

par le capteur situé entre les pièces A et B. Cette fois-ci l'évènement ne permet pas de savoir dans quel état le système est. Il nous faut maintenir en permanence l'état du système afin de savoir quelle transition franchir. En cas de problème d'observation il nous faut "spéculer" et considérer tous les cas possibles pondérés par des probabilités. L'état devient alors une densité de probabilité.

Pour comprendre comment un tel modèle fonctionne considérons l'observation suivante : « $\perp$, AD ». Le premier évènement reçu est $\perp$. Ce pseudo évènement signifie qu'un évènement a eu lieu, mais que l'on ne l'a pas observé. Si l'on veut pouvoir maintenir un état « spéculatif », il est nécessaire de savoir quand un évènement a été manqué.

Exécuter un modèle revient alors à déplacer des probabilités à la place de jeton. L'observation $\perp$ nous fait donc passer de l'état (a) indiquant que le robot a une probabilité 1 d'être dans la place A, à l'état (b) indiquant que le robot à une chance sur deux d'être dans la place B, et une chance sur deux d'être dans la place D.

L'observation de l'évènement AD n'est compatible qu'avec la présence du robot en place A et D. Le probabilité de 1/2 associé à la place D en (b) retourne dans la place A en (c) ; par contre la probabilité d'1/2 de la place B n'est pas compatible avec l'observation et doit être supprimée. On obtient alors l'état en (c), puis en normalisant les probabilité, celui en (d).

Cette petite exécution nous montre une des difficultés principales de cette approche : il est nécessaire de renormaliser les probabilité après suppression d'une possibilité afin d'obtenir (d). Sur un exemple aussi trivial, la normalisation est aisée ; elle se complique dans le cas général. Il n'est pas évident qu'une telle simplification soit faisable avec un moniteur à mémoire bornée.

7.2.3. D'observateur à chef d'orchestre

Une dernière piste consiste à enrichir le rôle de notre moniteur en lui faisant utiliser son observation pour agir sur le système. Cette enrichissement peut avoir deux utilisations possibles.

Actionneur L'idée est d'associer à certaines transitions (logiques ou évènementielles) un actionneur. Ainsi à chaque fois que la transition sera franchie, un actionneur sera enclenché. Ceci permet d'introduire des boucles de rétroactions, de déclencher des modes dégradés en cas d'erreur détectée, voire même d'implémenter des comportements plus complexes.

Injection de fautes Pour l'instant nous avons systématiquement utilisé notre moniteur pour surveiller un système en production. Cependant, un tel outil serait particulièrement adapté à l'injection de fautes. Le principe de l'injection de faute est comme son nom l'indique une pratique qui consiste à générer des entrées fautives pour voir comment le logiciel se comporte en cas d'entrée ne vérifiant pas les spécifications. L'intérêt d'un tel outil serait de générer des entrées à la volée selon le comportement du système étudié ; par exemple, on attend que le système atteigne un état particulier pour lancer une erreur.

Bibliographie

[ABLS05] Oliver Arafat, Andreas Bauer, Martin Leucker et Christian Schallhart. *Runtime verification revisited*. Rapport technique TUM-I0518. Technische Universität München, 2005.

[BCMG01] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann et Carl A. Gunter. « What packets may come : automata for network monitoring ». Dans : *ACM SIGPLAN Notices*. T. 36. 3. ACM. 2001, p. 206–219.

[BF12] Andreas Klaus Bauer et Yliès Falcone. « Decentralised LTL Monitoring ». Dans : *FM 2012 : Formal Methods-18th International Symposium*. T. 7436. 2012, p. 85–100.

[BG02] Karthikeyan Bhargavan et Carl A. Gunter. « Requirements for a practical network event recognition language ». Dans : *Electronic Notes in Theoretical Computer Science* 70.4 (2002), p. 1–20.

[BHFJ03] A. Benveniste, S. Haar, E. Fabre et C. Jard. « Distributed monitoring of concurrent and asynchronous systems ». Dans : *CONCUR 2003-Concurrency Theory* (2003), p. 1–26.

[BLS06] A. Bauer, M. Leucker et C. Schallhart. « Monitoring of real-time properties ». Dans : *FSTTCS 2006 : Foundations of Software Technology and Theoretical Computer Science* (2006), p. 260–272.

[BR08] M. Boyer et O. H. Roux. « On the compared expressiveness of arc, place and transition time Petri nets ». Dans : *Fundamenta Informaticae* 88.3 (2008), p. 225–249. issn : 0169-2968.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos et Sam Toueg. « The weakest failure detector for solving consensus ». Dans : *Journal of the ACM (JACM)* 43.4 (1996), p. 685–722.

[CJ05] T. Chatain et C. Jard. « Time supervision of concurrent systems using symbolic unfoldings of time Petri nets ». Dans : *Formal Modeling and Analysis of Timed Systems* (2005), p. 196–210.

[CR07] Feng Chen et Grigore Roşu. « Mop : an efficient and generic runtime verification framework ». Dans : *ACM SIGPLAN Notices*. T. 42. 10. ACM. 2007, p. 569–588.

[CT96] Tushar Deepak Chandra et Sam Toueg. « Unreliable failure detectors for reliable distributed systems ». Dans : *Journal of the ACM (JACM)* 43.2 (1996), p. 225–267.

[FA+03] Jean-Charles Fabre, Jean Arlat et al. « Building SWIFI tools from temporal logic specifications ». Dans : *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society. 2003, p. 95–95.

[FBHJ05] E. Fabre, A. Benveniste, S. Haar et C. Jard. « Distributed Monitoring of Concurrent and Asynchronous Systems ». Dans : *Discrete Event Dynamic Systems* 15.1 (2005), p. 33–84.

[FLP85] Michael J Fischer, Nancy A Lynch et Michael S Paterson. « Impossibility of distributed consensus with one faulty process ». Dans : *Journal of the ACM (JACM)* 32.2 (1985), p. 374–382.

[GP10] Alwyn E Goodloe et Lee Pike. *Monitoring distributed real-time systems : A survey and future directions*. National Aeronautics et Space Administration, Langley Research Center, 2010.

[Hav08] Klaus Havelund. *Runtime verification of C programs*. Springer, 2008.

[Hop91] R. Hopkins. « Distributable nets ». Dans : *Advances in Petri Nets 1991* (1991), p. 161–187.

[HR01] Klaus Havelund et Grigore Roşu. « Java PathExplorer–A runtime verification tool ». Dans : *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space : A New Space Odyssey*. 2001, p. 18–21.

[HR02] Klaus Havelund et Grigore Roşu. « Synthesizing monitors for safety properties ». Dans : *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002, p. 342–356.

[HR04] Klaus Havelund et Grigore Roşu. « An overview of the runtime verification tool Java PathExplorer ». Dans : *Formal methods in system design* 24.2 (2004), p. 189–215.

[JRR94] F. Jahanian, R. Rajkumar et S.C.V. Raju. « Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems ». Dans : *Real-Time Systems* 7.3 (1994), p. 247–273.

[KBPRR14] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange et Matthieu Roy. « Run-time Control to Increase Task Parallelism in Mixed-Critical Systems ». Dans : *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS14)*. 2014.

[KBPRRV13] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy et Fabian Vargas. « Monitoring On-line Timing Information to Support Mixed-Critical Workloads ». Dans : *Work in Progress Proceedings, IEEE RTSS 2013* (2013), p. 9–10.

[KVBAKLS99] Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee et Oleg Sokolsky. « Formally specified monitoring of temporal properties ». Dans : *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*. IEEE. 1999, p. 114–122.

[Lam01] Leslie Lamport. « Paxos made simple ». Dans : *ACM Sigact News* 32.4 (2001), p. 18–25.

[Lap+95] J.-C. Laprie et al. *Guide de la sûreté de fonctionnement*. Cépaduès-Éditions, Toulouse, 1995 - 1996.

[LMS02] François Laroussinie, Nicolas Markey et Philippe Schnoebelen. « Temporal logic with forgettable past ». Dans : *Logic in Computer Science, Symposium on*. IEEE Computer Society. 2002, p. 383–383.

[MR10] P. Meredith et G. Roşu. « Runtime Verification with the RV system ». Dans : *Proceedings of the First international conference on Runtime verification*. Springer-Verlag. 2010, p. 136–152.

[MSS93] Masoud Mansouri-Samani et Morris Sloman. « Monitoring distributed systems ». Dans : *Network, IEEE* 7.6 (1993), p. 20–30.

[PMCR08] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo et Grigore Roşu. « Hardware runtime monitoring for dependable cots-based real-time embedded systems ». Dans : *Real-Time Systems Symposium, 2008*. IEEE. 2008, p. 481–491.

[Ray10] M. Raynal. *Communication and Agreement Abstractions For Fault Tolerant Distributed Systems*. Morgan & Claypool, 2010.

[RFR08] T. Robert, J.C. Fabre et M. Roy. « On-line monitoring of real time applications for early error detection ». Dans : *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2008, p. 24–31.

[SVAR06] Koushik Sen, Abhay Vardhan, Gul Agha et Grigore Roşu. « Decentralized runtime analysis of multithreaded applications ». Dans : *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 2006, 11–pp.

[ZK10] Jun Zhu et Fabrice Kordon. « A Petri Net based runtime monitoring method for Web services specified with BPEL ». Dans : *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE. 2010, p. 304–310.

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

Table des figures

Table des définitions

Table des algorithmes

Table des matières