UQAC
Université du Québec
à Chicoutimi

MODEL CHECKING SUR DES PIPELINES DE STREAM PROCESSING

PAR ALEXIS BÉDARD

MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI COMME EXIGENCE PARTIELLE EN VUE DE L'OBTENTION DU GRADE DE MAÎTRE ÈS SCIENCES EN INFORMATIQUE

QUÉBEC, CANADA

RÉSUMÉ

L'*event stream processing* (ESP) est le traitement d'un flux continu d'objets, appelé séquence d'événements, dans l'optique de l'analyser ou de le transformer. Le laboratoire d'informatique formelle de l'UQAC (LIF) développe depuis plusieurs années un moteur de *stream processing open source* appelé BeepBeep 3. Cet engin permet une utilisation facile du concept d'ESP.

À BeepBeep 3, on y a intégré un système de vérification formelle. Avec seulement quelques lignes de code supplémentaires, il est maintenant possible de générer automatiquement une structure de Kripke d'une chaîne de processeurs donnée. Des applications intéressantes s'ajoutent donc à l'utilité déjà vaste de BeepBeep. Comparer des pipelines à l'aide de formules en logique temporelle linéaire (LTL) ou en logique du temps arborescent (CTL) et l'idée qu'une chaîne de processeurs monitore une structure de Kripke ne sont que quelques exemples.

Dans ce mémoire, on expliquera tout le processus de réflexion et d'exécution qui a mené à l'automatisation de la construction d'une chaîne de processeur BeepBeep en un modèle de Kripke valide pour analyse dans le logiciel NuXMV.

TABLE DES MATIÈRES

RÉSUMÉ ii
LISTE DES TABLEAUX vi
LISTE DES FIGURES viii
LISTE DES ABRÉVIATIONS 1
INTRODUCTION 2
CHAPITRE I – LES TYPES DE LOGIQUES MATHÉMATIQUES 9
1.1 LOGIQUE PROPOSITIONNELLE 9
1.1.1 SYSTÈME DE DÉDUCTION NATURELLE 10
1.1.2 COHÉRENCE ET COMPLÉTUDE 14
1.2 LOGIQUE DU PREMIER ORDRE 15
1.3 STRUCTURE DE KRIPKE 17
1.4 REPRÉSENTATION EN ARBRE 19
1.5 LOGIQUE TEMPORELLE LINÉAIRE (LTL) 20
1.6 LOGIQUE DU TEMPS ARBORESCENT (CTL) 23
1.6.1 DUALITÉ ET ÉQUIVALENCE CTL 28
1.7 AUTRES LANGAGES 28
CHAPITRE II – LA VÉRIFICATION PAR MODEL CHECKING 31
2.1 LES RÉSEAUX DE PETRI 32
2.1.1 RÉSEAUX DE PETRI GÉNÉRALISÉS 35
2.2 ALGORITHME CTL 38
2.3 ALGORITHME LTL 43
2.3.1 LES AUTOMATES DE BÜCHI 44
2.3.2 TRANSFORMATION D'UNE FORMULE LTL EN AUTOMATE 45
2.3.3 CONSTRUCTION DE L'AUTOMATE 47
2.3.4 COMBINAISON DES AUTOMATES 47

2.3.5 RECHERCHE D'UN CHEMIN 50
2.4 LOGICIELS DE *MODEL CHECKING* 50
CHAPITRE III – EVENT STREAM PROCESSING 56
3.1 COMPLEX EVENT PROCESSING 57
3.2 RUNTIME MONITORING 61
3.3 BEEPBEEP 3 66
CHAPITRE IV – MODÉLISATION DE PIPELINE EN NUXMV 74
4.1 DESCRIPTION DE LA SOLUTION 74
4.1.1 TRADUCTION 75
4.1.2 CONNEXION DES PROCESSEURS 77
4.1.3 LES EXCEPTIONS 80
4.1.4 DÉFI DES FILES D'ATTENTE 81
4.1.5 IMPLÉMENTATION JAVA 83
4.2 EXEMPLES D'APPLICATION 86
4.2.1 PROPRIÉTÉS D'UNE EXÉCUTION 87
4.2.2 MONITEUR DANS UN MONITEUR 87
4.2.3 VÉRIFICATION SUR LES FILES D'ATTENTE 88
4.2.4 COMPARAISON ENTRE DEUX PIPELINES 90
CHAPITRE V – ÉVALUATION EXPÉRIMENTALE 94
5.1 CONFIGURATION EXPÉRIMENTALE 94
5.2 RÉSULTATS ET DISCUSSION 97
5.2.1 IMPACT DES FILES D'ATTENTE 97
5.2.2 IMPACT DE LA TAILLE DU DOMAINE 99
5.2.3 IMPACT DE LA PROPRIÉTÉ 100
CONCLUSION 103
BIBLIOGRAPHIE 107
APPENDICE A – LISTE DE PROCESSEURS BEEPBEEP 3 POUVANT ÊTRE TRADUIT DANS NUXMV 115

APPENDICE B – REPRÉSENTATION VISUELLE DES PIPELINES SUR LESQUELLES NOUS AVONS EXÉCUTÉ NOS EXPÉRIENCES 118

APPENDICE C – RÉSULTATS EXPÉRIMENTAUX 120

LISTE DES TABLEAUX

TABLEAU 2.1 : TABLE DE VÉRITÉ DES SOUS-FORMULES DE $\varphi \mathbf{U} \psi$. 46

TABLEAU 2.2 : TABLE DÉMONTRANT LES ÉTATS VALIDES POUR $\neg\varphi = \varphi \mathbf{U} \psi$. 46

TABLEAU 5.1 : TEMPS DE VÉRIFICATION EN MILLISECONDES DES DIFFÉRENTES CHAÎNES DE PROCESSEURS POUR LES PROPRIÉTÉS *AUCUNE FILE PLEINE*, *LIVENESS* ET *BOUNDED LIVENESS* POUR UNE TAILLE DU DOMAINE DE 4 ET DES TAILLES DE FILES D'ATTENTE DE 3. 96

TABLEAU 5.2 : TEMPS EN MILLISECONDES DU PIPELINE PRODUCT OF 1 AND K-TH LORSQUE LA TAILLE DU DOMAINE EST 4. 98

TABLEAU 5.3 : TEMPS EN MILLISECONDES DU PIPELINE SIZE FOR SUM OF DOUBLES LORSQUE LA TAILLE DU DOMAINE EST 4. 98

TABLEAU 5.4 : NOMBRE D'ÉTATS DE LA STRUCTURE DE KRIPKE DU PIPELINE *PRODUCT OF 1 AND T-TH* SELON LES PARAMÈTRES Q_B ET N. 98

TABLEAU 5.5 : TEMPS EN MILLISECONDES DU PIPELINE PRODUCT OF WINDOW OF WIDTH 3 LORSQUE LA TAILLE DU DOMAINE EST 4. 99

TABLEAU 5.6 : TEMPS EN MILLISECONDES DU PIPELINE SUM OF 1S ON WINDOW LORSQUE LA TAILLE DU DOMAINE EST 4. 99

TABLEAU 5.7 : TEMPS DE VÉRIFICATION EN MILLISECONDES POUR LE PIPELINE SUM OF 1S ON WINDOW SELON LA LARGEUR DE LA FENÊTRE 100

TABLEAU 5.8 : TEMPS DE VÉRIFICATION EN MILLISECONDES POUR LE PIPELINE PRODUCT OF WINDOW OF WIDTH 3 SELON LA LARGEUR DE LA FENÊTRE 100

TABLEAU 5.9 : TEMPS EN MILLISECONDES DU PIPELINE PRODUCT OF 1 AND K-TH LORSQUE LA TAILLE DES FILES D'ATTENTE EST 3. 101

TABLEAU 5.10 : TEMPS EN MILLISECONDES DU PIPELINE SUM OF DOUBLE LORSQUE LA TAILLE DES FILES D'ATTENTE EST 3.. 101

TABLEAU 5.11 : TEMPS EN MILLISECONDES DU PIPELINE PRODUCT OF WINDOW OF WIDTH 3 LORSQUE LA TAILLE DES FILES D'ATTENTE EST 3. 101

TABLEAU 5.12 : TEMPS EN MILLISECONDES DU PIPELINE SUM OF 1S ON WINDOW LORSQUE LA TAILLE DES FILES D'ATTENTE EST 3. 101

TABLEAU 5.13 : TEMPS DE VÉRIFICATION EN MILLISECONDES POUR LA PROPRIÉTÉ ÉQUIVALENCE PAS-À-PAS. 102

TABLEAU 5.14 : TEMPS DE VÉRIFICATION EN MILLISECONDES POUR LA PROPRIÉTÉ ÉQUIVALENCE DE SÉQUENCE. 102

LISTE DES FIGURES

FIGURE 1.1 – PREUVE DU PRINCIPE DU TIERS EXCLU. EXEMPLE TIRÉ DU LIVRE DE HUTH ET RYAN À LA PAGE 25. 13

FIGURE 1.2 – UN EXEMPLE DE STRUCTURE DE KRIPKE. FIGURE SIMPLIFIÉE PROVENANT DE. 18

FIGURE 1.3 – REPRÉSENTATION ARBORESCENTE DE $(\neg P \vee Q) \rightarrow (R \wedge S)$. 19

FIGURE 1.4 – UN ARBRE OÙ E$\mathbf{F}\,\varphi$ EST SATISFAIT. 26

FIGURE 1.5 – UN ARBRE OÙ E$\mathbf{G}\,\varphi$ EST SATISFAIT. 26

FIGURE 1.6 – UN ARBRE OÙ A$\mathbf{G}\,\varphi$ EST SATISFAIT.. 27

FIGURE 1.7 – UN ARBRE OÙ A$\mathbf{F}\,\varphi$ EST SATISFAIT. 27

FIGURE 2.1 – UN EXEMPLE D'UN RÉSEAU DE PETRI. 33

FIGURE 2.2 – RÉSEAU DE PETRI DONT LE MARQUAGE INITIAL EST μ_0 = *(1, 0, 0, 0, 0)*. 33

FIGURE 2.3 – CHEMINS POTENTIELS DANS UN RÉSEAU DE PETRI. 34

FIGURE 2.4 – TROIS MOLÉCULES DE CHLORE ET DEUX MOLÉCULES DE PHOSPHORE DONNENT DEUX MOLÉCULES DE TRICHLORURE DE PHOSPHORE. 36

FIGURE 2.5 – UN SYSTÈME DE PRODUCTION-CONSOMMATION.. 37

FIGURE 2.6 – ÉTIQUETAGE DU MODÈLE POUR UNE FORMULE SOUS LA FORME $\exists\mathbf{X}\,\varphi$. 40

FIGURE 2.7 – ÉTIQUETAGE DU MODÈLE POUR UNE FORMULE SOUS LA FORME $\exists(\varphi\,\mathbf{U}\,\psi)$. 41

FIGURE 2.8 – ÉTIQUETAGE DU MODÈLE POUR UNE FORMULE SOUS LA FORME $\exists\mathbf{G}\,\varphi$. 41

FIGURE 2.9 – DIAGRAMME DE DÉCISION BINAIRE POUR LA FORMULE $Z_1 \wedge (\neg Z_2 \vee Z_3)$. 43

FIGURE 2.10 – EXEMPLE D'UN AUTOMATE DE BÜCHI DÉTERMINISTE. 45

FIGURE 2.11 – AUTOMATE FINAL DE LA FORMULE LTL $\neg\phi = \varphi \mathbf{U} \psi$. 48

FIGURE 2.12 – À GAUCHE, DEUX AUTOMATES. À DROITE, LE PRODUIT DES DEUX. 49

FIGURE 2.13 – EXEMPLE D'UN PROGRAMME À DEUX VARIABLES. 52

FIGURE 2.14 – MODÉLISATION DU PROGRAMME PRÉCÉDENT. 52

FIGURE 3.1 – UN EXEMPLE DE *QUANTIFIED EVENT AUTOMATA*, UTILISÉ DANS MARQ. 63

FIGURE 3.2 – UN EXEMPLE D'AUTOMATE, UTILISÉ DANS LARVA. 64

FIGURE 3.3 – UN EXEMPLE DE FORMULE, UTILISÉE DANS MONPOLY.. 65

FIGURE 3.4 – UNE SPÉCIFICATION JAVAMOP. 66

FIGURE 3.5 – REPRÉSENTATION GRAPHIQUE GÉNÉRIQUE D'UN PROCESSEUR PRENANT UN FLUX D'ÉVÉNEMENT EN ENTRÉE ET PRODUISANT UN FLUX DE SORTIE. 67

FIGURE 3.6 – PROCESSEURS DE BEEPBEEP 3.. 71

FIGURE 3.7 – PIPELINE METTANT EN VEDETTE LE PROCESSEUR FILTER. LES ÉVÉNEMENTS 6, 3 ET 8 SERONT PRODUITS EN SORTIE PAR CE PROCESSEUR. 71

FIGURE 3.8 – EXEMPLE DE PIPELINE CALCULANT LA SOMME À LA POSITION *I* ET *3I*. 72

FIGURE 4.1 – CODE DE DÉPART DE LA MODÉLISATION DU PROCESSEUR *COUNTDECIMATE*. 77

FIGURE 4.2 – DÉMONSTRATION DU MODULE MAIN DE L'EXEMPLE DU PIPELINE CALCULANT LA SOMME À LA POSITION I ET 3I. . . . 78

FIGURE 4.3 – SUITE DE LA MODÉLISATION DU PROCESSEUR *COUNTDECIMATE*. 79

FIGURE 4.4 – POUR QU'UNE ADDITION PRODUISE UNE VALEUR, LES DEUX ENTRÉES DOIVENT ÊTRE POURVUES D'AU MOINS UN ÉVÉNEMENT. 82

FIGURE 4.5 – EXEMPLE D'UTILISATION DE LA NOUVELLE EXTENSION NUXMV DE BEEPBEEP 3. 86

FIGURE 4.6 – STRUCTURE DE KRIPKE OBSERVÉE PAR UNE CHAINE DE PROCESSEURS. 89

FIGURE 4.7 – PIPELINE SURVEILLANT UNE PROPRIÉTÉ COMPLEXE. 89

FIGURE 4.8 – L'ÉQUIVALENCE PAS-À-PAS. 92

FIGURE 4.9 – L'ÉQUIVALENCE DE SÉQUENCE. 92

FIGURE 4.10 – PIPELINE QUI PRODUIT UNE SUITE DE NOMBRES CARRÉS. . . 92

FIGURE 4.11 – SIMPLIFICATION DU PIPELINE ORIGINAL. 93

FIGURE 4.12 – VERSION SIMPLIFIÉE POUR LES NOMBRES IMPAIRS. 93

FIGURE 5.1 – EXEMPLE DE CANAL AVEC PERTE. 106

LISTE DES ABRÉVIATIONS

CEP Complex event processing
CTL Computation Tree Logic
ENF Existential Normal Form
EPL Event Processing Language
ESP Event Stream Processing
GNBA Generalized Nondeterministic Büchi automaton
GPN Generalized Petri nets
LIF Laboratoire d'informatique formelle
LTL Linear Temporal Logic
NBA Nondeterministic Büchi Automaton
OBDD Ordered Binary Decision Diagram
RFID Radio Frequency Identification

INTRODUCTION

La quantité de données complexes et la vitesse à laquelle elles demandent d'être traitées sont aujourd'hui des problèmes que rencontrent nos applications. Dans ces cas-ci, ces applications doivent observer les événements et réagir au fur et à mesure qu'ils se produisent (Kleppmann, 2019). En ce sens, le *stream processing* est une technique permettant d'effectuer des calculs sur des flux d'événements. Un flux d'événements est défini par une suite d'unités appelée «événements» suivant un ordre prédéterminé (Luckham, 2005).

Dans le domaine du multimédia, plusieurs applications concrètes utilisent cette technique. Le traitement de signal, le rendu 2D et 3D et la compression audio et vidéo ne sont que quelques exemples (Owens *et al.*, 2002). Dans le domaine du matériel, il a été démontré comment le *stream processing* peut être implémenté avec un processeur FPGA (*field programmable gate arrrays*) pour mettre à profit un détecteur de mouvement (Serot *et al.*, 2011).

Avec les nouvelles technologies, de nouveaux domaines d'application ont émergé. Une application orientée recherche est une application qui utilise un moteur de recherche comme infrastructure principale pour rechercher et présenter des informations provenant de différentes sources de données. Un nouvel engin de *stream processing* (S4) a été développé pour résoudre des problèmes du monde réel dans le contexte de ces applications, où la quantité de données à traiter est gigantesque (Neumeyer *et al.*, 2010).

Monitorer des infrastructures ou des sites web est désormais commun ; un site peut souhaiter modéliser quels utilisateurs visitent une nouvelle page et un opérateur de service peut souhaiter surveiller les journaux afin de détecter les échecs en quelques secondes. Malheureusuement, les systèmes de *stream processing* existants ont une tolérance limitée face aux pannes et aux noeuds lents des graphes des réseaux sociaux. Des recherches se font afin de minimiser ces impacts (Zaharia *et al.*, 2013).

Le *stream processing* est un champ de recherche actif depuis plus de 20 ans et se développe aujourd'hui très rapidement grâce aux multitudes de logiciels *open source*. La notion de requêtes en continu a été introduite par le système Tapestry en 1992 (Terry *et al.*, 1992), puis suivie par de nombreuses recherches au début des années 2000.

Avec le temps, nous constatons que la technologie a bien évolué. Toutefois, le principe reste le même. Avec Tapestry, dans une base de données à laquelle des données sont continuellement ajoutées, les utilisateurs émettent une requête et sont avertis chaque fois que les données correspondent à celle-ci. Essentiellement, les concepts sont originaires de la communauté des bases de données et ont été implémentés dans des logiciels comme TelegraphCQ (Chandrasekaran *et al.*, 2003), construits pour résoudre le problème du flux continu de données provenant d'environnements en réseau, et Borealis/Aurora (Çetintemel *et al.*, 2016) dont les unités de traitement sont dotées de lignes de commande spéciales pouvant transporter de l'information permettant de changer le comportement de ces unités durant l'exécution.

Une deuxième génération de logiciels de *stream processing* est apparue lorsque l'intérêt a été déplacé vers des moteurs de traitement distribués et parallèles. On pense notamment à Storm et Apache Flink (Carbone *et al.*, 2015). Avec Storm, les événements sont des tuples clé-valeur. L'usager donne un identifiant unique au flux d'événements. Le flux est ensuite associé à un schéma fixe pour les tuples qu'il contient. Un projet Flink comprend trois types de processus : le client, le *JobManager* et au moins un *taskManager*. Le client prend le code du programme, le transforme en un graphique de flux de données et le soumet au *JobManager*. Le *JobManager* coordonne l'exécution du flux de données et le *taskManager* exécute au moins un opérateur qui produit un flux et rapporte leur statut au *taskManager*.

Pour ce projet, on a bénéficié du logiciel BeepBeep 3. BeepBeep 3 (Hallé, 2017) divise le calcul en unités simples, appelés processeurs. Les processeurs transforment une trace d'entrée

en une autre trace de sortie. Le résultat souhaité est obtenu en composant les sorties d'un processeur dans les entrées d'un autre, formant éventuellement une chaîne complexe. La première version de BeepBeep a été développée de 2008 à 2013. Cette version était bien plus limitée que la version actuelle et ne pouvait effectuer qu'un type spécifique de traitement de flux appelé *runtime monitoring*. La principale fonction de cette première version était la gestion des événements complexes avec une structure imbriquée comme les documents XML. BeepBeep 1 n'est plus en développement et est considéré comme obsolète. En 2013-2014, la version 2 était une tentative d'implémentation des mêmes concepts que ceux de BeepBeep 3.

Dans le domaine de l'informatique, la pression pour livrer des produits de qualité, dans les délais et qui répondent aux attentes des utilisateurs se fait sentir plus que jamais. Le manque de main-d'oeuvre et les coûts exorbitants peuvent être des facteurs, mais la difficulté à détecter et corriger des bogues accentue ce fléau. En effet, plus un bogue est décelé tard dans un projet, plus il risque de coûter cher (Austin & Williams, 2011). Heureusement, des techniques sont développées pour minimiser l'apparition de ses erreurs et faciliter leur découverte.

D'ailleurs, il est possible de détecter instantanément des bogues dans le domaine des systèmes de l'information à l'aide du *stream processing*, car des sources de données peuvent être vues comme une forme de flux d'événements. Par exemple, un capteur prenant des mesures de son environnement à intervalles périodiques produit un flux de valeurs numériques (Raghavan *et al.*, 2007). L'exécution d'un processus d'affaires progressant à travers une série d'actions peut être vue comme divers événements le long d'un enregistrement séquentiel (Koenig *et al.*, 2019). La plupart des systèmes d'information produisant des logs lors de leur exécution peuvent aussi être vus comme des sources de flux d'événements. Ces flux peuvent être trouvés dans des objets aussi divers que des entrées écrites dans un dossier médical (Berry & Milosevic, 2013) et même des flux des prix à la bourse (Behrend *et al.*, 2008).

D'autres techniques peuvent s'appliquer lors de la recherche de bogues ; elles s'appellent les méthodes formelles. Elles décrivent mathématiquement un système pour ensuite démontrer des propriétés sur sa structure et son fonctionnement (Batra, 2013). Les concepts mathématiques qui façonnent la base des méthodes formelles, sont notamment la logique propositionnelle (Klement, 2004) ainsi que la logique des prédicats (Smullyan, 1995). Ces deux types de notation permettent la modélisation de formules mathématiques d'un certain problème, rendant ainsi le raisonnement plus accessible. En bref, les méthodes formelles nous donnent les moyens nécessaires de vérifier qu'un système soit implémenté correctement ainsi que pour la plupart de vérifier ses propriétés sans avoir la nécessité de l'exécuter (Wing, 1990).

Les méthodes formelles se divisent en plusieurs branches les rendant ainsi utiles dans une multitude de situations. D'abord, nous avons les tests combinatoires qui identifient les interactions possibles entre des n-tuples de paramètres et consiste à tester toutes les paires de valeurs de variables d'entrée. Basé sur l'idée que si le comportement d'un système logiciel peut être affecté par un grand nombre de facteurs, alors seuls quelques facteurs sont impliqués dans un défaut provoquant une défaillance (Kacker *et al.*, 2013).

Le *runtime monitoring* est une technique de vérification qui regarde si un système sous surveillance satisfait la propriété de complétude (*correctness*) en analysant l'exécution courante et réagit si une anomalie est détectée (Francalanza *et al.*, 2017). C'est donc une technique efficace pour s'assurer qu'un système répond à un comportement désiré lors de l'exécution. Cette technique peut être utilisée dans de nombreux domaines d'application. ROSRV (Huang *et al.*, 2014) est un outil de *runtime verification* pour des applications robotiques. Ce logiciel vise à résoudre les problèmes de sûreté et de sécurité des robots en fournissant une infrastructure de surveillance qui intercepte les messages et surveille les messages transitant par le système. La NASA utilise le logiciel ROMR afin de vérifier des applications en langage C pour leurs missions spatiales. Un cas bien précis a été d'instaurer de la vérification sur un

système de fichier de sauvegarde dans le robot Curiosity lancé en 2011 sur Mars (Havelund, 2008).

De son côté, le *model checking* vérifie si une machine à états finis satisfait une certaine propriété. Si cette propriété n'est pas respectée, alors le logiciel, la majorité du temps, produit en sortie un contre-exemple représentant une exécution du modèle violant la propriété. La vérification doit prendre en compte toutes les exécutions possibles de la machine à états finis rendant ainsi le problème difficile. En étudiant un contre-exemple, il est facile de repérer l'erreur et de corriger le modèle par la suite. En revanche, si tout est conforme, le logiciel ne retourne qu'une confirmation. Par ces démarches, nous sommes capables d'accroître notre niveau de confiance envers nos modélisations (Baier & Katoen, 2008).

L'idée de départ qu'on voulait explorer était de savoir s'il était possible de vérifier des propriétés sur une chaîne de processeurs afin de s'assurer de sa validité. Ceci serait réalisé en exportant un pipeline BeepBeep dans un fichier d'entrée lisible par le logiciel de model checking NuXMV (Cimatti *et al.*, 2000). Après une recherche approfondie sur l'état de l'art, nous avons conclu que personne n'avait tenté de traduire les fonctionnalités d'un logiciel d'ESP en une structure de Kripke à des fins de vérification dans un *model checker*.

Le reste de ce mémoire se divisera ainsi : le chapitre 1 est consacré aux différentes formes de logique. On détaille la logique propositionnelle et donne des exemples de systèmes de déduction naturelle. On discute ensuite de la logique du premier ordre et des structures de Kripke. Des explications en profondeur sont données sur les langages CTL et LTL et ce chapitre se termine sur une énumération d'autres logiques temporelles.

Suit le chapitre 2 sur le *model checking*. Ce chapitre commence avec une section sur les réseaux de Petri, car ceux-ci ne peuvent être laissés sous silence quant à leur importance dans leurs apports sur le sujet. Des explications détaillées des algorithmes de *model checking*

CTL et LTL suivent. Ce chapitre se finit par une énumération des principaux logiciels de *model checking* actuellement sur le marché avec une plus grande attention portée sur NuXMV, puisque c'est celui-ci que nous avons utilisé pour notre projet.

Le chapitre 3 se consacre sur l'*event stream processing*. Une large portion de ce chapitre est accordée sur les engins d'ESP et de *runtime monitoring*. Finalement, nous détaillons les principaux processeurs BeepBeep. Un processeur BeepBeep est une pièce de construction fondamentale au logiciel. Cet objet prend un ou plusieurs événements en entrée et retourne un ou plusieurs événements en sortie.

Le chapitre 4 explique la traduction dans NuXMV des pipelines BeepBeep 3. Nous élucidons comment nous avons réussi à transformer un pipeline en structure de Kripke. Après avoir énoncé les concepts, nous plongeons dans le code Java pour en expliquer les détails de l'implémentation. On parle des défis rencontrés, comme la modélisation des files d'attente et des processeurs *Window* et *GroupProcessor*. Aussi, ce chapitre montre des applications qu'apporte notre contribution. Avec l'apport du *model checking*, nous pouvons maintenant comparer deux chaînes de processeurs entres elles. Par exemple, nous pouvons nous assurer que deux chaînes de processeurs sont équivalentes et produisent les mêmes résultats bien qu'elles sont différentes. De plus, nous pouvons faire des vérifications sur les listes d'attente des processeurs dont nous expliquerons les concepts plus tard. Un dernier avantage de notre modélisation est de monitorer une structure de Kripke à l'aide d'un pipeline BeepBeep.

Le chapitre 5 énonce nos résultats expérimentaux. On étudie les facteurs influençant le temps d'exécution et la complexité de la vérification ; nous identifions les processeurs nécessitant le plus de ressources et de mémoire.

Nous concluons en réitérant le potentiel de notre développement expérimental, mais nous expliquons la fragilité qu'apporte une augmentation des files d'attente. L'utilisation du

logiciel NuXMV pour nos modélisations a aussi amené son lot de défi. Une avenue intéressante serait de modéliser les mêmes chaînes de processeurs dans le logiciel SPIN et d'en comparer les résultats. SPIN permettrait l'utilisation de canaux de communication entre les processeurs. Ceci réduirait peut-être la complexité et l'impact qu'ont les listes d'attente sur le temps de vérification. Enfin, comme travaux futurs, nous nous questionnons sur les effets qu'aurait une modélisation de pipelines avec pertes de données, comme cela peut survenir dans les environnements réseau.

Le travail présenté dans ce mémoire a fait l'objet de deux publications : la première a été présentée à la conférence TIME 2021 (28^{th} International Symposium on Temporal Representation and Reasoning) et s'intitule «Model Checking of Stream Processing Pipelines» (Bédard & Hallé, 2021). Pour sa part, la deuxième a été publiée dans le journal scientifique *Information and Computation* et porte le titre «Formal verification for event stream processing : Model checking of BeepBeep stream processing pipelines» (Bédard & Hallé, 2023).

CHAPITRE I

LES TYPES DE LOGIQUES MATHÉMATIQUES

La logique est au coeur de plusieurs domaines d'études : dans le domaine du langage (Allwood *et al.*, 1977), de la philosophie (Kern *et al.*, 1983) ainsi que les mathématiques (Visser, 1981). Peu importe le domaine d'étude, les travaux effectués en logique permettent de faciliter la modélisation mathématique d'un problème et facilitant sa résolution.

Dans ce chapitre, on entre dans les détails de la logique propositionnelle et de la logique du premier ordre. Les explications et les exemples sont largement inspirés du livre *Logic in Computer Science*, écrit par Michael Huth et Mark Ryan (Huth & Ryan, 2004).

1.1 LOGIQUE PROPOSITIONNELLE

D'abord, une proposition en logique propositionnelle peut être évaluée soit par vrai ($\top$), soit par faux ($\bot$). Par exemple, en français, nous pourrions affirmer que la phrase « les oiseaux volent » est une formule propositionnelle, car cette phrase déclarative peut être évaluée à $\top$. En revanche, une phrase telle que «Ferme la porte» n'en est pas une, car elle ne peut être ni vraie ni fausse.

En logique propositionnelle, on remplace les énoncés textuels par des littéraux pour faciliter la représentation. Dans notre exemple ci-haut, « les oiseaux volent » pourrait être remplacé par la lettre p. Nous pouvons ensuite combiner des littéraux afin de construire des expressions plus complexes, appelées clauses, avec l'aide des connecteurs logiques. Ces symboles représentent le *ou*, le *et*, la négation ainsi que *si ... alors*.

Définition 1.1.1. *Les expressions dites bien formées (EBF) en logique propositionnelle sont celles obtenues en utilisant les règles de construction ci-dessous :*

p, q : Un littéral est toujours bien formé.

$\vee$: La disjonction $\varphi \vee \psi$, se lisant φ ou ψ, est vraie exactement lorsqu'au moins un des deux opérandes φ et ψ est vrai.

$\wedge$: La conjonction $\varphi \wedge \psi$, se lisant φ *et* ψ, est vraie exactement lorsque φ et ψ sont vraies.

$\rightarrow$: L'implication $\varphi \rightarrow \psi$, se lisant *si* φ *alors* ψ, est une conséquence logique de φ. La seule situation où cette proposition est fausse est lorsque que φ est vraie, mais que ψ est fausse.

$\neg$: La négation $\neg\varphi$, se lisant non φ, indique la valeur inverse de la formule.

Ces définitions nous permettent récursivement de déterminer si toute EBF est vraie ou fausse à partir de la valeur de vérité de ses littéraux. Les expressions suivantes sont des formules propositionnelles bien formées :

$$p \rightarrow (r \vee q) \qquad p \wedge q \wedge r$$

En revanche, les expressions suivantes ne sont pas des formules propositionnelles bien formées. Pour celle de gauche, une expression ressemblant à $p \wedge q$ ou encore $p \rightarrow q$ serait valide ; deux connecteurs ne peuvent être un à côté de l'autre. Pour celle de droite, $p \wedge r$ et $q \wedge r$ seraient des clauses bien formées. Deux littéraux ou formules doivent être raccordés par un connecteur :

$$p \rightarrow \wedge q \qquad pq \wedge r$$

1.1.1 SYSTÈME DE DÉDUCTION NATURELLE

Poursuivons l'étude de la logique propositionnelle en greffant à ce qu'on connait le système de déduction naturelle. Ce système de preuve introduit les régles d'inférence pour

des fins de démonstration. Celles-ci fondent le processus de démonstration et l'application de ces règles permettent d'en démontrer les théorèmes. Il existe deux types de règles : des règles d'introduction et des règles d'élimination. Simplement, dans leur forme générale, les règles d'introduction introduisent un symbole logique à l'étape de la conclusion, alors que les règles d'élimination les éliminent pour n'en conclure qu'un littéral ou une expression.

L'écriture de ces règles prend généralement la même forme. Au dessus de la ligne se trouvent les prémisses de la régle. Une prémisse est une affirmation dont on tire une conclusion. En dessous de la ligne va ladite conclusion. À droite, on retrouve entre parenthèses le nom de la règle. Le nom commence toujours par le connecteur logique que nous considérons. Ce symbole est ensuite suivi soit par un e si la règle élimine le symbole logique soit par un i si celle-ci en introduit un.

La règle d'introduction de la conjonction ($\wedge i$) permet de prouver $\varphi \wedge \psi$ si nous avons déjà conclu φ et ψ séparément. De son côté, la règle d'élimination ($\wedge e$) peut conclure φ ou ψ si $\varphi \wedge \psi$ est déjà prouvé.

$$\frac{\varphi \qquad \psi}{\varphi \wedge \psi}\,(\wedge i) \qquad \frac{\varphi \wedge \psi}{\varphi}\,(\wedge e_1) \quad \frac{\varphi \wedge \psi}{\varphi}\,(\wedge e_2)$$

Les règles de disjonction stipulent qu'on doit assumer soit φ ou ψ pour prouver $\varphi \vee \psi$. Pour que cette expression soit vraie, nous savons qu'elle n'a besoin que de l'un ou l'autre et c'est pourquoi il existe deux règles d'introduction.

$$\frac{\varphi}{\varphi \vee \psi}\,(\vee i_1) \quad \frac{\psi}{\varphi \vee \psi}\,(\vee i_2)$$

La règle d'élimination affirme qu'afin de prouver une expression χ à partir d'une disjonction, alors χ doit être démontrable à partir de φ ainsi que ψ. Puisque nous ne savons pas lequel de φ et ψ est vraie, nous devons donner deux preuves distinctes dont nous avons combiné en une seule.

$$\frac{\varphi \vee \psi \qquad \begin{array}{c}\varphi \\ \vdots \\ \chi\end{array} \qquad \begin{array}{c}\psi \\ \vdots \\ \chi\end{array}}{\chi} (\vee e)$$

La règle d'introduction de l'implication stipule que si on peut conclure ψ en partant de φ, alors on peut en conclure $\varphi \to \psi$. La règle d'élimination affirme que si on sait φ et $\varphi \to \psi$, alors on peut conclure ψ. Aussi connu sous le nom de *Modus ponens*, elle peut être lu comme suit : «Si φ est vraie et que $\varphi \to \psi$ l'est aussi, alors ψ doit être vraie».

$$\frac{\begin{array}{c}\varphi \\ \vdots \\ \psi\end{array}}{\varphi \to \psi} (\to i) \qquad \frac{\varphi \qquad \varphi \to \psi}{\psi} (\to e)$$

Finalement, la règle d'introduction de la négation affirme qu'une supposition φ menant à une contradiction donne sa négation $\neg\varphi$. L'élimination de la négation assure que s'il est possible de conclure à la fois φ et $\neg\varphi$, alors il y a forcément contradiction.

$$\frac{\begin{array}{c}\varphi \\ \vdots \\ \bot\end{array}}{\neg\varphi} (\neg i) \qquad \frac{\varphi \qquad \neg\varphi}{\bot} (\neg e)$$

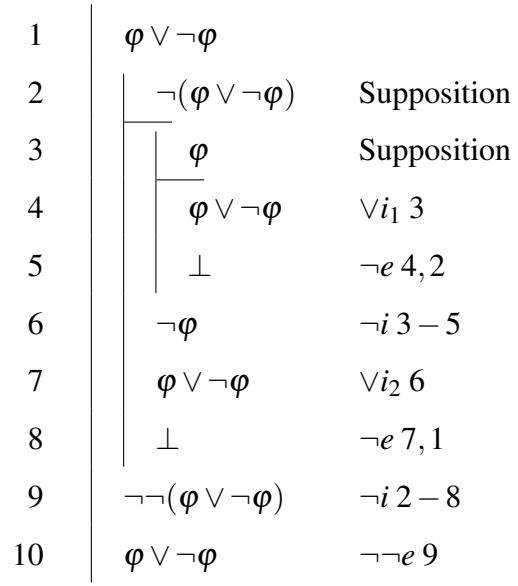


FIGURE 1.1 : Preuve du principe du tiers exclu. Exemple tiré du livre de Huth et Ryan à la page 25.

Ces règles nous permettent maintenant de raisonner sur des problèmes. La déduction naturelle est le processus d'aboutir à une conclusion ψ à partir de prémisses φ_1, φ_2, ..., φ_n en utilisant les règles ci-dessus.

Appliquons-les dans un exemple complet : Prouvons le principe du tiers exclu ($\varphi \vee \neg\varphi$). Ce principe cite qu'il est impossible d'avoir une chose et son contraire. Ce principe de non-contradiction stipule que pour toute proposition φ, on ne peut pas avoir φ et $\neg\varphi$ en même temps (Barzin & Errera, 1929). Attention : Cette affirmation ne doit pas être confondue avec le principe du même nom formulé par Aristote.

Analysons la preuve ligne par ligne. Nous supposons d'abord l'inverse de la prémisse, soit $\neg(\varphi \vee \neg\varphi)$. En supposant ensuite φ, à l'aide de la règle d'introduction de la disjonction $\vee_1$, il est possible de conclure $\varphi \vee \neg\varphi$. Nous avons donc une contradiction (ligne ¬ e 4,2). Les

lignes six, sept et huit sont semblables aux précédentes, mais avec $\neg\varphi$ comme supposition. Encore une fois, nous arrivons à une contradiction. Par le fait même, nous pouvons conclure l'inverse de notre première supposition, donc $\neg\neg(\varphi \vee \neg\varphi)$.

Cet exemple montre qu'en appliquant les règles en succession, on peut en déduire une conclusion à partir d'un ensemble de prémisses. Dans l'exemple ci-haut, notre ensemble de prémisses était un ensemble vide. Supposons qu'on a un ensemble de prémisses φ_1, φ_2, ..., φ_n prouvant une conclusion ψ, nous pouvons dénoter le résultat par φ_1, φ_2, ..., $\varphi_n \vdash \psi$. Puisque dans l'exemple nous n'avions aucune prémisse, alors le résultat est seulement $\vdash (\varphi \vee \neg\varphi)$.

1.1.2 COHÉRENCE ET COMPLÉTUDE

Maintenant les bases maîtrisées, introduisons deux propriétés importantes de la logique propositionnelle, soit la cohérence et la complétude. Soit φ_1, φ_2, ..., φ_n un ensemble de prémisses et ψ une conclusion. La notation φ_1, φ_2, ..., $\varphi_n \vdash \psi$ indique qu'il est possible de déduire ψ à partir des prémisses ($\models \psi$). L'implication sémantique $\models$ affirme que ψ sera toujours $\top$ si chacune de prémisses sont vraies. En d'autres mots, il n'existe aucune conclusion qui se révèle fausse, alors que les prémisses sont vraies.

Posons-nous la question : Est-ce que $\varphi \wedge \psi \models \varphi$ est une affirmation valide ? Ici, nous devons inspecter chacune des évaluations possibles de φ et de ψ. Lorsque ceux-ci donnent $\top$ pour $\varphi \wedge \psi$, il faut s'assurer que φ le soit aussi. Comme on le voit dans la table de vérité, $\varphi \wedge \psi$ est vrai seulement lorsque φ et ψ le sont tous les deux. Donc, $\varphi \wedge \psi \models \varphi$ est effectivement vrai.

ψ	φ	$\psi \wedge \varphi$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

Pouvons-nous en dire autant de $\varphi \vee \psi \models \varphi$? Dans ce cas, il existe trois combinaisons pour que $\varphi \vee \psi$ soit vraie, dont celle où φ est $\bot$ et ψ est $\top$. Donc $\varphi \vee \psi \models \varphi$ ne tient pas.

La complétude s'explique par le fait que si φ_1, φ_2,..., $\varphi_n \models \psi$ est valide, alors il existe une preuve de déduction naturelle pour la séquence φ_1, φ_2,..., $\varphi_n \vdash \psi$. Combiné avec le résultat du théorème de cohérence, on obtient φ_1, φ_2,..., $\varphi_n \vdash \psi$ est valide si et seulement si φ_1, φ_2,..., $\varphi_n \models \psi$ l'est également.

1.2 LOGIQUE DU PREMIER ORDRE

La logique propositionnelle comporte à elle seule de grandes limitations. Comme nous l'avons vu, elle permet de traiter les énoncés ne prenant en compte que les connecteurs *non, et, ou* et *si ... alors*. Cependant, que faire si nous voulons utiliser des mots comme *tous* et *il existe* ? C'est la raison pour laquelle nous introduisons le quantificateur universel $\forall$ qui signifie «pour tout» et $\exists$ qui signifie «il existe». Une règle importante s'applique à ces quantificateurs : ceux-ci doivent toujours être rattachés à une variable. Donc, le quantificateur $\forall x$ se lit «pour tout x» et $\exists y$ se lit «il existe un y».

La logique du premier ordre, aussi appelé la logique des prédicats, permet l'utilisation de ces quantificateurs. Elle autorise les relations et dépendances parmi les individus et permet leur généralisation. Mathématiquement, un prédicat est une fonction *f*, prenant un nombre

déterminé d'objets d'un ensemble E selon l'arité du prédicat. Un prédicat doit toujours retourner $\top$ ou $\bot$ selon les arguments qu'on lui passe.

Par exemple, considérons l'affirmation suivante :

Ce ne sont pas tous les oiseaux qui peuvent voler.

Nous pourrions tenter de traduire cette phrase à l'aide de la logique propositionnelle, mais nous ne saurons en garder le sens. Avec ce que nous connaissons à l'heure actuelle, il est impossible de généraliser sur un groupe de population ni d'en isoler une certaine partie.

Dans ce cas-ci, nous pourrions décider de modéliser cette affirmation au moyen de deux prédicats O et V possédant tous les deux un seul argument :

O(x) : x est un oiseau
V(x) : x peut voler

Nous pouvons maintenant exprimer correctement l'expression à l'aide des prédicats de cette façon :

$$\neg(\forall x(O(x) \rightarrow V(x)))$$

Puisque les prédicats O et V ne comportent qu'un seul argument, on dira ici qu'ils sont d'arité un. Forcément, un prédicat peut avoir plusieurs paramètres. Si un en contient plusieurs, on dira que ce prédicat est une relation. Par exemple, une arité de deux sera représentée $J(x,y)$. Dans ce cas-ci, disons que le prédicat J signifie «plus jeune que», alors on dit que x est « plus jeune que» y.

Comparativement à la logique propositionnelle, la validité des prédicats est fortement liée à son environnement. En effet, sans contexte, il est impossible d'évaluer $J(x,y)$. Au fond, les variables ne sont que des caractères génériques en vue de les substituer dans un contexte précis. Une variable liée est une variable à l'intérieur de la portée d'un quantificateur $\forall x$ ou $\exists x$. Considérons la formule suivante :

$$\varphi = (\forall x\ (P(x) \land Q(x))) \rightarrow (\neg P(x) \lor Q(y))$$

Ici, les variables liées au quantificateur $\forall x$ sont la variable x dans les prédicats P(x) et Q(x) à l'intérieur de la première parenthèse. Toutes variables se trouvant à l'extérieur de celle-ci sont libres.

1.3 STRUCTURE DE KRIPKE

Les logiques précédentes permettent de formuler des énoncés dans un monde *statique* : chaque proposition ou prédicat est vrai ou faux et ces valeurs sont immuables. Dans ce qui suit, on s'intéressera aux systèmes dynamiques. Une structure de Kripke est un système de transitions utilisé pour le *model checking* (Chapitre 2). Un système est formé d'un ensemble d'états ; dans chaque état, des propositions peuvent être soit vraies ou fausses, et le système évolue en passant d'un état à un autre, formant ainsi une exécution. En général, plusieurs exécutions sont possibles à partir du même état de départ. Un système de transitions, qu'on appellera dorénavant modèle, est un ensemble d'états noté S, une relation de transitions entre les états notée R et une fonction d'étiquetage L. La fonction d'étiquetage indique quels sont les littéraux qui sont vrais à l'intérieur des états. Dans son livre *Model Checking*, Edmund Clarke le définit ainsi (Clarke *et al.*, 1999) :

Définition 1.3.1. *Un modèle $\mathscr{M}$ = (S, R, L) est défini par :*

— *S : l'ensemble des états* $s \in S$

— *R : est la relation de transitions* $R \subseteq S \times S$. $(s, s') \in S$ *s'il existe une transition de l'état s vers* s'

— *L : une fonction de marquage des états* $L : S \rightarrow 2^{AP}$, *où AP est l'ensemble des propositions atomiques.*

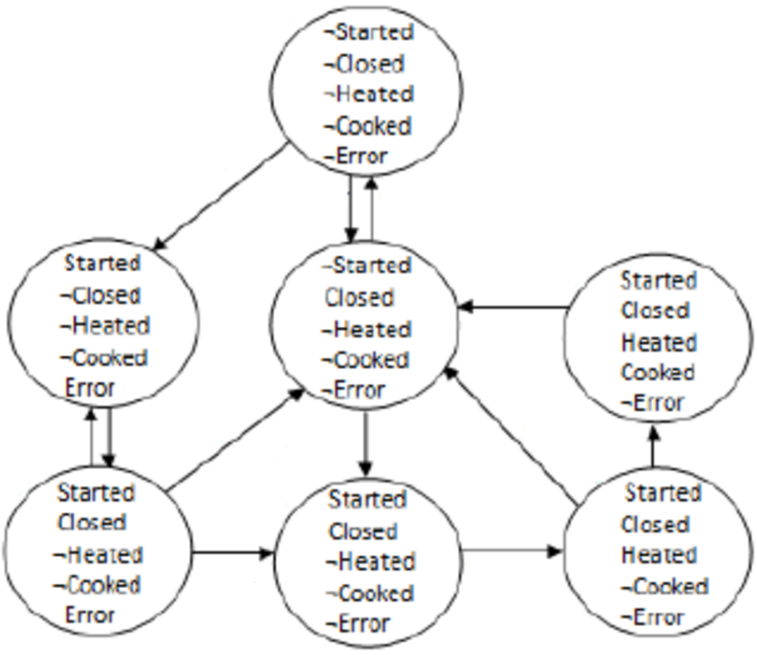


FIGURE 1.2 : Un exemple de structure de Kripke. Figure simplifiée provenant de (De Menezes *et al.*, 2010)

Dans la représentation d'une structure de Kripke, les états sont représentés par des cercles. Tous les cercles ont une ou plusieurs transitions orientées vers le même ou un autre état. À l'intérieur des états, la fonction de marquage identifie les variables du problème en inscrivant celles qui sont vraies et fausses. Celles qui sont fausses sont accompagnées par un tilde dans l'exemple illustré à la figure 1.2 qui représente simplement les états d'un micro-

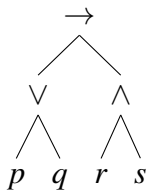


FIGURE 1.3 : Représentation arborescente de $(\neg p \vee q) \rightarrow (r \wedge s)$.

ondes. On dit qu'un chemin $\bar{s}$ est une suite d'états s_1, s_2, ..., s_i d'un modèle respectant le système de transitions. Un chemin est toujours infini.

1.4 REPRÉSENTATION EN ARBRE

Les formules de logique peuvent être représentées de manière arborescente. Être en mesure de visualiser rapidement une formule de cette façon peut être un avantage dans certaines situations, notamment pour bien comprendre le fonctionnement des algorithmes récursifs de modèle checking comme nous le verrons plus tard.

Voici un exemple de construction d'un arbre syntaxique de la formule $(\neg p \vee q) \rightarrow (r \wedge s)$. À la racine de l'arbre se trouve l'opérateur principal de la formule. Par conséquent, $\rightarrow$ est à la racine. Puisque notre formule est divisée en deux termes, $\varphi = \neg p \vee q$ et $\psi = (r \wedge s)$, notre arbre sera binaire. Le processus est répété jusqu'aux feuilles de l'arbre. Le côté gauche peut être visualisé comme un sous-arbre et ainsi on recherche encore une fois le connecteur principal de φ qui est le connecteur $\vee$. Les littéraux p et q complètent ce côté de l'arbre. Du côté droit, le connecteur $\wedge$ est à la racine du sous-arbre et les feuilles sont les littéraux r et s. La figure 1.3 modélise la représentation arborescente de la formule.

1.5 LOGIQUE TEMPORELLE LINÉAIRE (LTL)

La logique temporelle linéaire (en anglais *Linear temporal logic ou LTL*) est une logique temporelle permettant de modéliser des propriétés sur des séquences en se référant au futur. En fait, pour se rapporter au futur en LTL, il existe six mots-clés importants à connaître. Nous les appelons des opérateurs.

1. Next (**X**)
2. Eventually (**F**)
3. Globally (**G**)
4. Until (**U**)
5. Weak-until (**W**)
6. Release (**R**)

Définition 1.5.1. *LTL est défini sous la syntaxe suivante, respectant la forme Backus Naur :*

$$\varphi ::= \mid \bot \mid \top \mid p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \boldsymbol{X}\varphi \mid \boldsymbol{F}\varphi \mid \boldsymbol{G}\varphi \mid (\varphi\ \boldsymbol{U}\varphi) \mid (\varphi\ \boldsymbol{W}\varphi) \mid (\varphi\ \boldsymbol{R}\varphi)$$

Définition 1.5.2. *Soit $\mathscr{M}$ = (S, R, L) une structure de Kripke et $\bar{s} = s_1, s_2, \ldots$ un chemin dans $\mathscr{M}$. $\bar{s}$ satisfait une formule LTL et est défini par la relation $\vDash$ si :*

1. $\bar{s} \vDash p$ si et seulement si p appartient au marquage de l'état actuel $L(s)$
2. $\bar{s} \vDash \neg\varphi$ si et seulement si $\bar{s} \nvDash \varphi$
3. $\bar{s} \vDash \varphi \wedge \psi$ si et seulement si $\bar{s} \vDash \varphi$ et $\bar{s} \vDash \psi$
4. $\bar{s} \vDash \varphi \vee \psi$ si et seulement si $\bar{s} \vDash \varphi$ ou $\bar{s} \vDash \psi$
5. $\bar{s} \vDash \varphi \rightarrow \psi$ si et seulement si $\bar{s} \vDash \psi$ lorsque $\bar{s} \vDash \varphi$

6. $\bar{s} \vDash \mathbf{X}\varphi$ si et seulement si $\bar{s}^2 \vDash \varphi$.

7. $\bar{s} \vDash \mathbf{F}\varphi$ si et seulement s'il existe un $i \geq 1$ tel que $\bar{s}^i \vDash \varphi$.

8. $\bar{s} \vDash \mathbf{G}\varphi$ si et seulement si, pour tout $i \geq 1$, $\mathrm{s}^i \vDash \varphi$.

9. $\bar{s} \vDash \varphi \mathbf{U} \psi$ si et seulement s'il existe un $i \geq 1$ tel que tel que $\bar{s}^i \vDash \psi$ et pour tout $j = 1, \ldots, i-1$, on a $\mathrm{s}^j \vDash \varphi$.

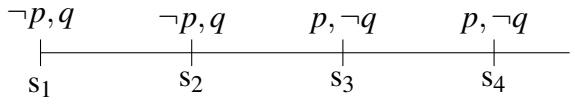
10. $\bar{s} \vDash \varphi \mathbf{W} \psi$ si et seulement si soit il existe un $i \geq 1$ tel que $\bar{s}^i \vDash \psi$ et pour tout $j = 1, \ldots, i-1$ on a $\bar{s}^j \vDash \varphi$ ou pour tout $k \geq 1$ on a $\bar{s}^k \vDash \varphi$.

11. $\bar{s} \vDash \varphi \mathbf{R} \psi$ si et seulement si soit il existe $i \geq 1$ tel que $\bar{s}^i \vDash \varphi$ et pour tout $j = 1, \ldots, i$ $\mathrm{s}^j \vDash \psi$ ou pour tout $k \geq 1$ on a $\bar{s}^k \vDash \psi$.

Les définitions 1 à 5 correspondent à ce que nous avons vu en logique propositionnelle. La sixième définition affirme que l'opérateur **X** retire le premier état du chemin et que ce nouveau point de départ doit satisfaire la formule φ. La septième stipule qu'éventuellement, un suffixe dans le chemin satisfera φ. À l'inverse, pour la huitième, tous les suffixes du chemin doivent la satisfaire. L'opérateur Until stipule que la condition φ doit tenir jusqu'à ce que ψ soit satisfaite. Également, le Weak-Until ressemble beaucoup au Until, mais rajoute la condition que si ψ ne tient jamais, alors φ doit rester $\top$. Finalement, le Release ressemble aussi au Until, mais le changement de valeurs des conditions doivent se faire dans le même état. Tout comme le weak-until, φ doit rester $\top$ si ψ reste $\bot$ définitivement.

Considérons maintenant un exemple. Soit le chemin infini $\bar{s}$ suivant et s_1 le point de départ :

Exemple 1.5.1.



Commençons par l'opérateur $\mathbf{X}$. Selon l'exemple ci-dessus, il serait valide d'écrire $\mathbf{X}(\neg p)$, puisqu'à l'état s_2, nous avons bel et bien $\neg p$. Ensuite, $\mathbf{F}p$ se révèle vrai, puisqu'en s_3, p devient $\top$. Pour $\mathbf{G}$, on pourrait affirmer $\mathbf{G}(p \vee q)$. Par la suite, il serait admissible d'écrire $\neg p \,\mathbf{U}\, \neg q$ et $\neg p \,\mathbf{W}\, \neg q$. Le littéral p reste effectivement négatif jusqu'à ce qu'en s_3, q devienne aussi négatif. Finalement, l'expression $\neg p \,\mathbf{R}\, \neg q$ retourne $\bot$, puisqu'en s_3, là où le littéral q change de valeur, le littéral p, en revanche, avait déjà changé d'évaluation.

On a expliqué comment évaluer des formules LTL sur une séquence unique. Si on a une structure de Kripke, on peut étendre cette définition et affirmer qu'une structure $\mathscr{M}$ satisfait une formule si toutes ses exécutions satisfont la formule. En reprenant l'exemple du micro-ondes à la figure 1.2, on peut se convaincre que la formule $\neg \mathit{heat}\ \mathbf{U}\, \mathit{closed}$, qui affirme que le micro-ondes n'est jamais en fonction jusqu'à ce que la porte ne soit fermée, est vraie peu importe le chemin qu'on suit.

La notion d'équivalence signifie que φ et ψ sont interchangeables. On dit que deux formules LTL φ et ψ sont équivalentes et l'on note $\varphi \equiv \psi$, si pour tous les chemins $\bar{s}$ on a : $\bar{s} \vDash \varphi$ si et seulement si $\bar{s} \vDash \psi$. Dans une formule LTL, si φ s'y trouve et que $\varphi \equiv \psi$, alors on peut substituer φ par ψ. Suite à cette idée, nous pouvons affirmer que $\mathbf{F}$ et $\mathbf{G}$ sont duals l'un de l'autre, alors que $\mathbf{X}$ est dual à lui-même. En effet, pour toute expression φ, nous pouvons montrer que

$$\neg\, \mathbf{G}\, \varphi \equiv \mathbf{F}\, \varphi \qquad \neg\, \mathbf{F}\, \varphi \equiv \mathbf{G}\, \neg \varphi \qquad \neg \mathbf{X}\, \varphi \equiv \mathbf{X}\, \neg \varphi$$

Tout comme $\mathbf{F}$ et $\mathbf{G}$, $\mathbf{U}$ et $\mathbf{R}$ sont duals l'un de l'autre, car

$$\neg(\varphi\ \mathbf{U}\, \psi) \equiv \neg \varphi\ \mathbf{R}\, \neg \psi \qquad \neg(\varphi\ \mathbf{R}\, \psi) \equiv \neg \varphi\, \mathbf{U}\, \neg \psi$$

Puisque Release et Weak-until sont très similaires, il est possible d'en créer des équivalences :

$$\varphi \ \mathbf{W}\, \psi \equiv \psi \ \mathbf{R}\, (\varphi \vee \psi) \quad \varphi \ \mathbf{R}\, \psi \equiv \psi \mathbf{W}\, (\varphi \vee \psi)$$

En connaissant ces équivalences, notons qu'il n'est plus nécessaire d'utiliser les opérateurs Release et Weak-Until. Tout d'abord, il est possible d'écrire le Weak-Until en disjonction d'Until et Globally.

$$\varphi \ \mathbf{W}\, \psi \equiv \varphi \mathbf{U}\, (\psi \vee \mathbf{G}\, \psi)$$

Finalement, le Release peut simplement s'écrire à l'aide du connecteur Until, de la négation des formules ψ et φ, puis de la négation du tout.

$$\varphi \ \mathbf{R}\, \psi \equiv \neg(\neg\varphi \ \mathbf{U} \neg\psi)$$

1.6 LOGIQUE DU TEMPS ARBORESCENT (CTL)

La section précédente expliquait que la logique LTL ne pouvait être évaluée que sur un chemin. En revanche, la logique du temps arborescent (en anglais *computation tree logic* ou CTL), est une logique dite arborescente. Cette logique considère toutes les exécutions d'une structure de Kripke, à partir d'un état initial. Chaque exécution possible correspond à une traversée d'un arbre à partir de la racine.

Rappelons-nous que la section 1.2 introduisait les quantificateurs $\forall$x et $\exists$x. Tout comme ces symboles enrichissaient la logique propositionnelle, ils complètent la logique LTL rajoutant ainsi la possibilité de raisonner sur les chemins en plus des propriétés. Cependant, en CTL,

les quantificateurs s'utilisent de façon différente qu'en logique du premier ordre. En effet, le symbole $\forall$ signifie «tous les chemins» et $\exists$ signifie «il existe un chemin».

Définition 1.6.1. *Nous définissons la syntaxe CTL de la façon suivante :*
$\varphi ::= \bot \mid \top\, p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \forall X\, \varphi \mid \exists X\, \varphi \mid \forall \boldsymbol{F}\, \varphi \mid \exists \boldsymbol{F}\, \varphi \mid \forall \boldsymbol{G}\, \varphi \mid \exists \boldsymbol{G}\, \varphi \mid \forall [\varphi\ \boldsymbol{U}\, \psi] \mid \exists [\varphi\ \boldsymbol{U}\, \psi]$

Définition 1.6.2. *Soit $\mathscr{M}$ = (S, R, L) et $\bar{s} = s_1, s_2, \ldots$ un chemin dans $\mathscr{M}$. Un état satisfait une formule CTL et est défini par la relation $\vDash$ si :*

1. $\mathscr{M}, s \vDash p$ si et seulement si p appartient au marquage de l'état actuel $L(s)$
2. $\mathscr{M}, s \vDash \neg\varphi$ si et seulement si φ
3. $\mathscr{M}, s \vDash \varphi \wedge \psi$ si et seulement si $\mathscr{M}, s \vDash \varphi$ et $\mathscr{M}, s \vDash \psi$
4. $\mathscr{M}, s \vDash \varphi \vee \psi$ si et seulement si $\mathscr{M}, s \vDash \varphi$ ou $\mathscr{M}, s \vDash \psi$
5. $\mathscr{M}, s \vDash \varphi \rightarrow \psi$ si et seulement si $\mathscr{M}, s \nvDash \varphi$ ou $\mathscr{M}, s \vDash \psi$
6. $\mathscr{M}, s \vDash \forall \mathrm{X}\varphi$ si et seulement si pour tous les prochains états s$\rightarrow$s$_1$ on a $\mathscr{M}$, $s_1 \vDash \varphi$.
7. $\mathscr{M}, s \vDash \exists \mathrm{X}\varphi$ si et seulement s'il existe un prochain état tel que $s \rightarrow$s$_1$ on a $\mathscr{M}$, $s_1 \vDash \varphi$.
8. $\mathscr{M}, s \vDash \forall \mathrm{G}\varphi$ si et seulement si pour tous les chemins $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où $s_1 = s$, et pour tous les s$_i$ tout au long du chemin, on a $\mathscr{M}$, $s_i \vDash \varphi$.
9. $\mathscr{M}, s \vDash \exists \mathrm{G}\varphi$ si et seulement s'il existe un chemin $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où s_1 est égal s, et pour tous les s$_i$ tout au long du chemin, on a $\mathscr{M}$, $s_1 \vDash \varphi$.
10. $\mathscr{M}, s \vDash \forall \mathrm{F}\varphi$ si et seulement si pour tous les chemins $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où s_1 est égal s, il existe un s$_i$ tels que $\mathscr{M}$, $s_i \vDash \varphi$.
11. $\mathscr{M}, s \vDash \exists \mathrm{F}\varphi$ si et seulement s'il existe un chemin $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où s_1 est égal à s, et pour quelque s$_i$ tout au long du chemin, $\mathscr{M}$, $s_i \vDash \varphi$.

12. $\mathscr{M}, s \vDash \forall[\varphi_1 \mathrm{U} \varphi_2]$ si et seulement si tous les chemins $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où s_1 est égal à s, ce chemin satisfait $\varphi_1 \mathrm{U} \varphi_2$. Il existe un s_i tout au long du chemin, tel que $\mathscr{M}$, $s_i \vDash \varphi_2$, et pour tous j $\leq$ i, on a $\mathscr{M}$, $s_j \vDash \varphi_1$.

13. $\mathscr{M}, s \vDash \exists[\varphi_1 \mathrm{U} \varphi_2]$ si et seulement s'il existe un chemin $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ où s_1 est égal s, ce chemin satisfait $\varphi_1 \mathrm{U} \varphi_2$. Il existe des s_i tout au long du chemin, tel que $\mathscr{M}$, $s_i \vDash \varphi_2$, et pour tous j $\leq$ i, on a $\mathscr{M}$, $s_j \vDash \varphi_1$.

Les définitions 1 à 5 correspondent à ce que nous avons vu en logique propositionnelle. La sixième se lit «tous les prochains états» satisfont une formule φ, tandis que la septième se lit « φ est vraie pour au moins un possible prochain état». Ensuite, la définition numéro huit se traduirait par «pour tous les chemins, la propriété tiendra globalement». En revanche, la suivante irait en ce sens : «Il existe un chemin où la propriété tiendra globalement». Les deux prochaines ressemblent aux deux précédemment citées, mais il faut remplacer les mots «la propriété tiendra globalement» par «la propriété tiendra éventuellement». En d'autres mots, elle deviendra $\top$ à un moment ou un autre. Finalement, les deux dernières signifient «pour tous les chemins, φ tient jusqu'à ce que ψ tiennent» et «il existe un chemin tel que φ tient jusqu'à ce que ψ tiennent».

Ces dernières définitions réfèrent aux chemins des modèles. Il serait donc utile de visualiser avec des exemples.

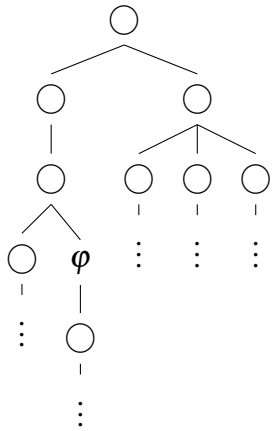


FIGURE 1.4 : Un arbre où EF φ est satisfait.

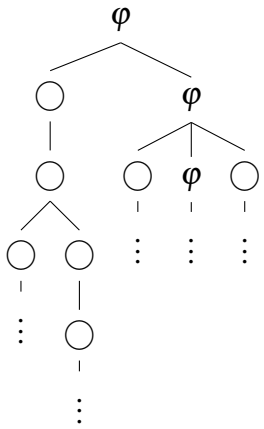


FIGURE 1.5 : Un arbre où EG φ est satisfait.

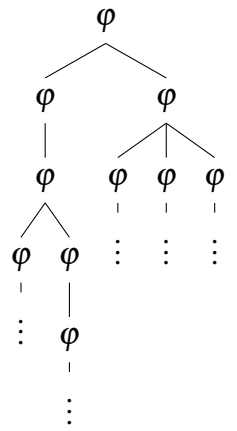


FIGURE 1.6 : Un arbre où AG φ est satisfait.

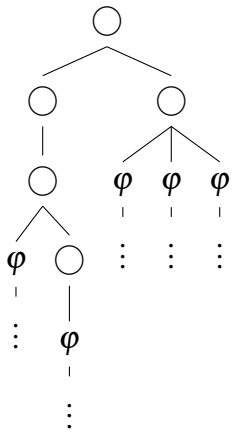


FIGURE 1.7 : Un arbre où AF φ est satisfait.

La figure 1.4 est la visualisation arborescente de la spécification «Il existe un chemin où éventuellement, φ sera $\top$». Ensuite, la figure 1.5 réfère à «Il existe un chemin où globalement, φ est $\top$». Pour ce qui est de la figure 1.6, celle-ci signifie «Tous les chemins ont globalement φ évaluée à $\top$». Finalement, la figure 1.7 énonce que «Tous les chemins auront éventuellement la propriété φ évaluée à $\top$».

1.6.1 DUALITÉ ET ÉQUIVALENCE CTL

Similairement à LTL, il existe des équivalences parmi les opérateurs CTL.

Définition 1.6.3. *On dit que deux formules CTL φ et ψ sont sémantiquement équivalentes $\varphi \equiv \psi$, si tout état dans tout modèle qui satisfait l'un d'eux satisfait également l'autre.*

En effet, les formules CTL sont duals l'un l'autre s'ils sont de signe opposé ainsi que leur contenu. Par exemple, $\forall$ et $\exists$ sont opposés et **F** et **G** le sont aussi. L'exemple ci-dessous montre deux formules CTL sémantiquement équivalentes :

$$\forall \mathbf{F}\, \varphi \equiv \exists \mathbf{G}\, \neg \varphi$$

1.7 AUTRES LANGAGES

Pour conclure ce chapitre, mentionnons qu'il existe d'autres langages de logiques temporelles outre CTL et LTL. Le langage CTL*, qui propose un mélange entre ces derniers, est un des plus expressifs. En effet, CTL* laisse tomber la contrainte des opérateurs devant absolument être associés à des quantificateurs. De plus, CTL* permet la sélection d'un intervalle sur un chemin comme le permet LTL. Cette logique donne la possibilité d'écrire des formules telles que :

— $\forall[(p\,\mathbf{U}\,r) \vee (q\,\mathbf{U}\,r)]$ *Dans tous les chemins, soit $p\,\mathbf{U}\,r$ ou $q\,\mathbf{U}\,r$.*

— $\forall[\mathbf{X}\,p \vee \mathbf{X}\,\mathbf{X}\,p]$ *Dans tous les chemins, p est vraie au prochain état ou p est vraie dans le deuxième prochain.*

Il existe un sous-ensemble de la logique LTL qui s'appelle LTL_{-x}. En fait, LTL_{-x} est l'utilisation de LTL sans l'opérateur **X**. Cette logique est surtout utilisée dans les modèles concurrents et les exécutions asynchrones (Karaman, 2009).

L'introduction de la «Metric Temporal Logic» par Ron Koymans (Koymans, 1990), était justifiée par le besoin d'un langage de spécification couvrant les systèmes à temps réel. En effet, bien que LTL et LTL_{-x} ne soient définis que sur l'ensemble des nombres entiers naturels $\mathbb{N}$, MTL introduit les espaces de temps où les transitions se produisent. Le modèle de séquence qu'on a vues peut être formalisé comme une association entre les nombres naturels (positifs) et l'ensemble AP : est associé au nombre 1 l'ensemble des littéraux qui sont vrais dans le premier état, au nombre 2 ceux du deuxième état, et ainsi de suite. MTL généralise cette notion en associant plûtot les nombres rationnels à AP. C'est donc dire que cette logique est valide sur l'ensemble des nombres rationnel positif $\mathbb{Q}^+$ et que MTL permet d'être plus précis dans les contraintes de temps. Par l'introduction d'une variable δ, où δ est une fonction de distance *d(t, t')* donnant la mesure par rapport au temps de la distance entre *t* et *t'*, nous pouvons mentionner qu'une proposition p deviendra vraie dans une demie unité de temps. Ce δ, s'il est présent, doit toujours suivre un opérateur MTL. Par exemple, ce qu'on pouvait représenter en LTL par $\mathbf{G}(p \rightarrow \mathbf{F}q)$, peut maintenant être décrit en insérant une contrainte de temps avec $\mathbf{G}(p \rightarrow \mathbf{F}_{<0.5}q)$. Cette formule signifie «si p est rencontré, alors q dans un temps maximal de 0.5 unité de temps, et ce, globalement».

Une extension de MTL a été développée pour y intégrer des intervalles de temps. Celle si s'appelle la «metric interval temporal logic» (MITL). Avec cette logique, il est possible de mentionner qu'une variable *p* deviendra vraie dans un intervalle de deux à dix unités de temps (Alur *et al.*, 1996). L'intervalle remplace la variable δ de MTL. Par exemple, $\mathbf{G}(p \rightarrow \mathbf{F}_{[2,10]}q)$ représente ce qui vient d'être mentionné.

Finalement, le «signal temporal logic» (STL) est adapté pour vérifier des propriétés sur des signaux continus (Maler & Nickovic, 2004). Assumons des signaux x1[t], x2[t],..., xn[t], alors les predicats seront de la forme f(x1[t],x2[t], . . . , xn[t]) > 0 où f est une fonction quelconque calculée à partir des valeurs de chaque signal de temps t. En évaluant un signal

donné, il est possible pour le prédicat de retourner $\top$ ou $\bot$, comme la définition l'impose. Ainsi, la formule STL $\mathbf{G}\left((x1 > 0.7) \rightarrow \mathbf{F}_{[0,0.5]}(x2 > 0.7)\right)$ vérifie que la valeur de x_2 dépasse effectivement 0.7 au plus tard 0.5 unité de temps après que celle de x_1 ait, elle aussi, dépassée 0.7 tout au long du signal.

CHAPITRE II
LA VÉRIFICATION PAR MODEL CHECKING

Les raisons pour vérifier la justesse d'un système ou d'un matériel informatique sont nombreuses. Par exemple, il est primordial de tester en profondeur les systèmes critiques à la sécurité (Lahtinen *et al.*, 2012), les systèmes dans la production de masse (Fages, 2003), etc.

Ce chapitre concerne une méthode de vérification appelée la vérification de modèles, ou *model checking* en anglais. La vérification de modèle est basée sur la logique temporelle, telle que vue dans le chapitre précédent. Elle vient avec une sémantique qui définit précisément quand une expression est vraie ou fausse pour une trace ou une expression donnée. Ainsi, la notion de vérité est remplacée par une notion plus dynamique selon laquelle les formules peuvent changer leur valeur de vérité à mesure que le système évolue.

Le model checking commence avec un modèle décrit pas l'utilisateur et cherche si les hypothèses émises par ce dernier sont valides sur ce modèle. Si elles ne le sont pas, un contre-exemple est émis par le système en démontrant une trace d'exécution. Le model checking se concentre exclusivement sur des propriétés temporelles.

Les définitions sémantiques de LTL et CTL nous permettent de tester les états d'un modèle afin de savoir s'ils satisfont une formule LTL ou CTL. En général, les systèmes de transition auront un très grand nombre d'états et la formule peut, elle aussi, être très longue. Il vaut donc d'essayer de trouver des algorithmes efficaces.

2.1 LES RÉSEAUX DE PETRI

Les réseaux de Petri ont été inventés par Carl Adams Petri alors qu'il n'avait que 13 ans. Il les employait pour visualiser les processus de transformation des molécules (Petri & Reisig, 2008). Aujourd'hui, ils sont un outil indispensable dans la modélisation de systèmes de calcul concurrent asynchrone.

Les réseaux de Petri sont, jusqu'à un certain point, très intéressant pour notre travail puisque des travaux ont démontré que du model checking peut être fait sur ceux-ci ce qui est semblable à notre recherche (Latvala & Mäkelä, 2004).

Les réseaux de Petri sont des graphes orientés dont les noeuds sont appelés des places et les flèches des arcs. Des lignes verticales séparent les places entres-elles et déterminent les transitions. Formellement, un réseau de Petri C est défini selon le quadruplet $C = (P, T, I, O)$, où P est un ensemble de places et T un ensemble de transitions. La fonction d'entrée I indique, pour chacune des transitions $t_i \in T$, l'ensemble des places d'entrée auxquelles elle est associée. La fonction de sortie O définit l'ensemble des places de sorties (Peterson, 1977).

L'image 2.1 montre un réseau de Petri à cinq places et quatre transitions. Chacune des composantes $C = (P, T, I, O)$ est donnée par :

— $P = \{p_1, p_2, p_3, p_4, p_5\}$

— $T = \{t_1, t_2, t_3\}$

— $I(t_1) = \{p_1\} \quad O(t_1) = \{p_2, p_3\}$

— $I(t_2) = \{p_2, p_3\} \quad O(t_2) = \{p_4\}$

— $I(t_3) = \{p_4, p_5\} \quad O(t_3) = \{p_5\}$

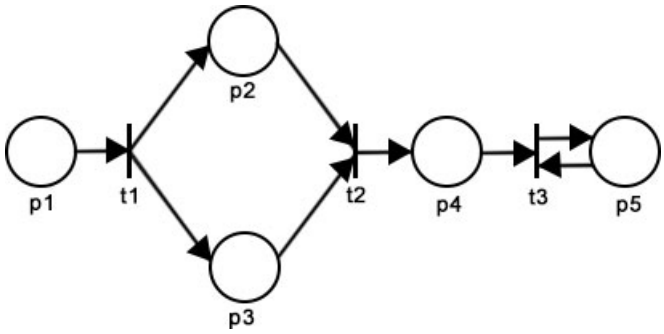


FIGURE 2.1 : Un exemple d'un réseau de Petri.

Une place peut contenir un ou plusieurs jetons représentés par des points noirs. Les arcs montrent les chemins que ces jetons peuvent suivre dans le réseau. Un jeton permet le marquage du réseau à un instant. Un réseau de Petri C demande un marquage initial μ_0 démontrant les jetons dans les places à l'état initial. La définition d'un réseau de Petri marqué M devient donc $M = (P, T, I, O, \mu)$. $\mu = (\mu_0, \mu_1, \mu_2, ..., \mu_n)$ est un vecteur donnant, pour chaque place dans le réseau, le nombre de jeton à cet endroit. Par exemple, pour un réseau de Petri à trois places, le vecteur $\mu_i = (1, 0, 2)$ signifie qu'à cet instant, un jeton se trouve dans la place p_1, aucun dans p_2 et deux dans p_3. La figure 2.2 représente un réseau de Petri à 5 places dont le marquage initial est $\mu_0 = (1, 0, 0, 0, 0)$.

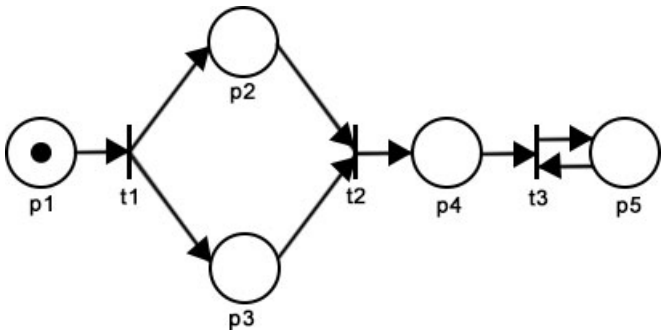


FIGURE 2.2 : Réseau de Petri dont le marquage initial est $\mu_0 = (1, 0, 0, 0, 0)$.

La figure 2.3 montre les quatre types de transitions possibles dans un réseau de Petri marqué. D'abord, nous avons la transition séquentielle simplement formée d'une place en entrée et une autre en sortie. Un jeton sera consommé pour en produire un en sortie. La seconde alternative est un chemin conflictuel, c'est-à-dire un jeton en entrée ayant le choix entre deux sorties. Cette décision est non-déterministe. La troisième possibilité est une transition concurrente. Une transition concurrente est une entrée alimentant deux sorties. Dans ce cas, un jeton en entrée crée un jeton dans les deux sorties. Enfin, la quatrième transition possible est la synchrone, étant deux entrées n'alimentant qu'une seule place en sortie. Il est essentiel d'y avoir au moins un jeton dans les deux entrées pour créer un et un seul jeton en sortie.

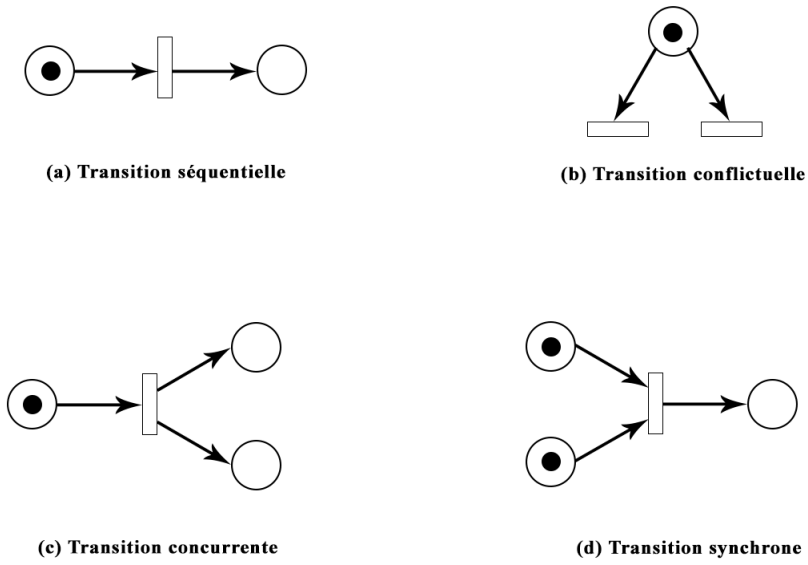


FIGURE 2.3 : Chemins potentiels dans un réseau de Petri.(Wang, 1998)

Avec ces règles, on peut analyser le scénario de la figure 2.2. Comme vu précédemment, la transition t_1 est concurrente. À partir de p_1, puisqu'il y a au moins un jeton dans toutes les places en entrée, le jeton sera consommé et un jeton sera créé dans la place p_2 et un autre

dans p_3. Ensuite, p_2 et p_3 sont synchrones. Puisqu'il y a un jeton dans ces places, alors un jeton sera créé dans p_4. De p_4, la transition est séquentielle et lorsque le jeton se trouve dans p_5, il ne reste qu'une transition séquentielle vers elle-même et le scénario se termine avec un marquage $\mu_3 = (0, 0, 0, 0, 1)$.

2.1.1 RÉSEAUX DE PETRI GÉNÉRALISÉS

Les réseaux de Petri généralisés (en anglais *generalized Petri nets* ou GPN) lèvent la contrainte qu'une place peut lancer ou recevoir qu'un seul jeton à la fois. On a expliqué qu'une place d'entrée est reliée par un arc à une transition et que cette transition est aussi reliée à une place de sortie par un arc. Avec les GPN, une place peut être reliée par plusieurs arcs à une transition, signifiant que cette place doit contenir au moins un nombre égal de jetons que d'arcs afin que la transition puisse s'effectuer. Ce nombre de jetons sera donc consommé à la transition.

Les réseaux de Petri généralisés sont d'excellents moyens pour illustrer des équations chimiques. Prenons l'équation suivante :

$$3Cl_2 + 2P \rightarrow 2PCl_3$$

Cette formule affirme que trois molécules de chlore (Cl_2) et deux molécule de phosphore (P) peuvent s'unir par des liaisons chimiques et former deux molécules de trichlorure de phosphore ($2PCl_3$). La figure 2.4 est le réseau de Petri généralisé de cette formule chimique.

D'autres variants de réseaux de Petri existent, notamment les réseaux de Petri temporisés, qui introduisent une notion de temps dans les transitions des jetons, ainsi que les réseaux de Petri stochastiques qui rajoutent un délai aléatoire à ces transitions.

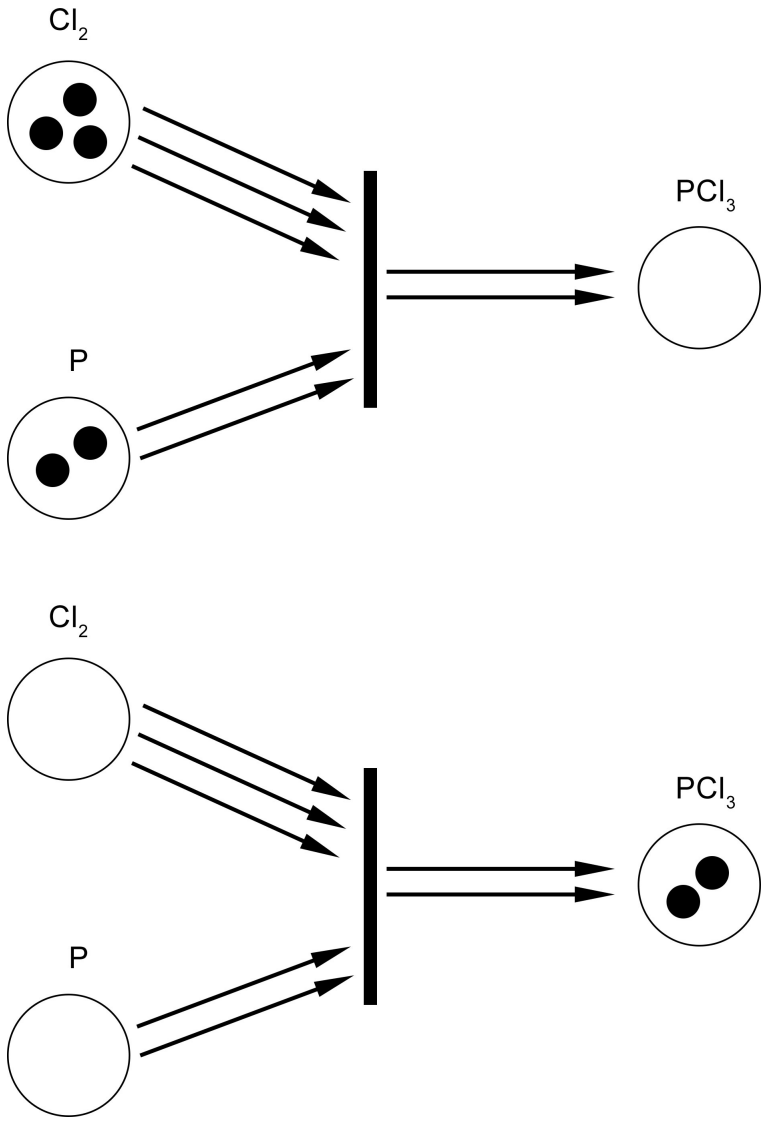


FIGURE 2.4 : Trois molécules de chlore et deux molécules de phosphore donnent deux molécules de trichlorure de phosphore.

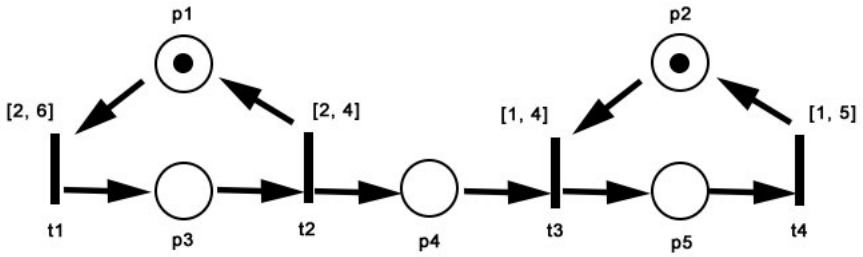


FIGURE 2.5 : Un système de production-consommation.

RÉSEAUX DE PETRI TEMPORISÉS

Une variante qu'on va explorer est réseau de Petri temporisé (timed Petri net en anglais ou TPN). Contrairement aux réseaux de Petri réguliers qui n'ont aucune condition quant aux transitions des jetons, les TPNs introduisent deux valeurs de temps. Les arcs seront maqués par un intervalle de αs à βs. En effet, αs est le temps minimum que la transition doit attendre après qu'elle soit activée et avant qu'elle soit lancée. Pour βs, on parle plutôt du maximum de temps que la transition peut attendre avant de se déclencher.

Un réseau de Petri temporisé est donc un sextuplet TPN = (P, T, I, O, μ, SI), où (P, T, I, O, μ) est un réseau de Petri marqué et SI une représentation appellée intervalle statique T $\rightarrow$ Q^* s ($Q^* \cup \infty$) et où Q^* est l'ensemble des nombres rationnels positifs (Wang, 2012).

Dans un TPN, un intervalle associé à une transition détermine la durée du déclenchement. Pendant cet d'intervalle, un jeton n'est ni dans une place d'entrée ni de sortie.

Exemple : la figure 2.5 représente un système de deux nœuds ; un producteur et un consommateur. Le producteur prend 2 à 6 unités de temps pour produire un produit, puis prend 2 à 4 unités de temps pour le rendre au consommateur. Le consommateur prend 1 à 4 unités de temps pour saisir le produit, et 1 à 5 unités de temps pour le consommer.

Les TPN sont notamment utilisés dans l'analyse de systèmes concurrents asynchrones (Ramchandani, 1973), la modélisation de systèmes complexes en temps réel (van der Aalst, 1993) et l'étude d'un modèle d'ordonnancement pour un séquencement optimal de la production dans un système d'assemblage flexible (Zhang *et al.*, 2005).

2.2 ALGORITHME CTL

Le *model checking* est une technique de vérification automatique sur des modèles finis. Voyons d'abord l'algorithme de vérification CTL. L'algorithme de vérification CTL s'agit d'un algorithme de *model checking* qui consiste à déterminer si une formule CTL est vraie sur une structure de Kripke donnée. Il consiste en les étapes suivantes : par un système d'étiquetage, la procédure réunit d'abord dans un ensemble tous les états d'un modèle dont la formule ϕ est vraie et s'assure par la suite que tous les états initiaux appartiennent à cet ensemble.

L'algorithme effectue les opérations d'étiquetage que sur des formules CTL sous la forme normale existentielle, ou ENF (Existential Normal Form). Cette forme conserve toute l'expressivité de CTL, mais en ne permettant que les quantificateurs existentiels. La forme ENF ne considèrera que les formules ϕ sous les formes p, $\varphi \wedge \psi$, $\exists X \varphi$, $\exists(\varphi U \psi)$ ou $\exists G \varphi$.

Une première étape consiste donc à modifier toutes formules CTL en ENF. Grâce à la dualité et la négation, on peut vérifier que les équivalences suivantes permettent l'élimination des quantificateurs universels. Toutes les formules peuvent être réécrites en forme normale existentielle. Toutefois, la traduction peut causer une explosion exponentielle en raison de la règle $\forall(\varphi \text{ U } \psi)$.

— $\forall X \varphi \equiv \neg \exists X \neg \varphi$

— $\forall(\varphi \text{ U } \psi) \equiv \neg \exists(\neg \psi \text{ U } (\neg \varphi \wedge \neg \psi)) \wedge \neg \exists G \neg \psi$

— $\forall \text{F} \varphi \equiv \neg \text{s G} \neg \varphi$

— $\forall G\varphi \equiv \neg s\ (\top\ U\ \varphi)$

L'algorithme CTL consiste en un marquage des états selon la formule ϕ. D'abord, on la décompose en sous-formules jusqu'aux propositions atomiques. On marque les états avec les propositions atomiques qui sont vraies dans ces états. On «remonte» en marquant les états avec des sous-formules progressivement plus complexes, en se basant sur le marquage des formules plus simples dont elles sont composées. Les techniques de marquage diffèrent toutefois selon les opérateurs.

— p : on marque par un littéral tous les états dont ce littéral est présent.

— $\varphi \wedge \psi$: On marque les états par $\varphi \wedge \psi$ si φ et ψ sont présents.

— $\exists \mathbf{X}\, \varphi$: on marque par $\exists \mathbf{X}\, \varphi$ tous les états avec qui ont une transition vers un état marqué par φ.

— $\exists(\varphi\, \mathbf{U}\, \psi)$: on marque les états qui sont la source d'un chemin atteignant un état marqué par ψ où tous les états précédents sont marqués par φ.

— $\exists \mathbf{G}\, \varphi$: on marque par $\exists \mathbf{G}\, \varphi$ les états dont les chemins sont précédés par des états marqués par φ.

Les figures 2.6, 2.7 et 2.8 sont des exemples de marquage pour $\exists \mathbf{X}\, \varphi$, $\exists(\varphi\, \mathbf{U}\, \psi)$ et $\exists \mathbf{G}\, \varphi$ respectivement. Dans la première, φ est vraie dans les états s_3 et s_4, alors que ψ est vraie seulement dans s_1. Les états de ce modèle où $\exists \mathbf{X}\, \varphi$ est vraie sont les états s_2, s_3 et s_4.

Dans la deuxième, puisque φ est vraie dans les états s_3 et s_4 et que ψ l'est dans s_1, alors la formule $\exists(\varphi\, \mathbf{U}\, \psi)$ est vraie dans les états s_1, s_3 et s_4. En effet, seul s_1 ne satisfait pas la propriété, car ni φ ni ψ est vraie.

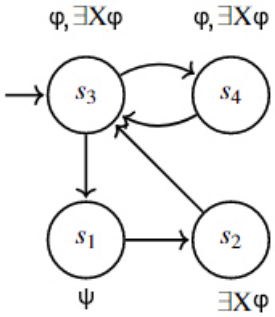


FIGURE 2.6 : Étiquetage du modèle pour une formule sous la forme $\exists \mathbf{X}\, \varphi$.

La troisième nécessite de définir la notion de composantes fortement connexe : une composante fortement connexe se définit par un sous-graphe, tel que pour toute paire de sommets u et v, il existe un chemin de u vers v et de v vers u.

Pour cet étiquetage, nous devons chercher les chemins où chaque noeud est marqué avec φ et étiqueter les états de ces chemins avec l'étiquette $\exists G\varphi$. L'algorithme est répété en prenant comme point de départ chacun des états satisfaisants φ. L'état noté par s_1 ne sera pas considéré. Il ne reste donc que les états s_2, s_3 et s_4 à évaluer. Parmi ceux-ci, on trouve ceux qui appartiennent à une composante fortement connexes du graphe restant et on marque ces états avec $\exists \mathbf{G}\, \varphi$. Dans notre cas, les états s_3 et s_4 forment une composante fortement connexe. Finalement, puisque tous les états qui ne sont pas marqués par φ ont été ignorés à la première étape, il ne reste qu'à marquer les états restants du graphe.

Enfin, pour qu'une formule CTL ϕ soit satisfaite dans un modèle, celle-ci doit être vraie sur chaque chemin partant des états initiaux. C'est-à-dire qu'on doit déterminer si les états initiaux font partis de l'ensemble des états satisfaisant ϕ. Si on reprend les figures 2.6, 2.7, puisque s_3 est l'ensemble des états initiaux et que ceux-ci sont étiquetés par leur formule ϕ respective, alors l'algorithme CTL conclut que la formule est vraie. En revanche, dans

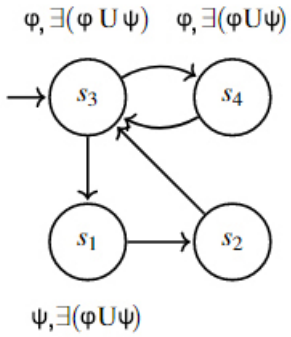


FIGURE 2.7 : Étiquetage du modèle pour une formule sous la forme $\exists(\varphi \, \mathrm{U} \, \psi)$.

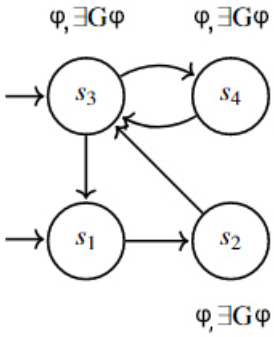


FIGURE 2.8 : Étiquetage du modèle pour une formule sous la forme $\exists \mathrm{G} \, \varphi$.

l'exemple de la figure 2.8, l'ensemble des états initiaux est s_1, s_3, mais s_3 n'est pas étiqueté par $\exists \mathbf{G}\, \varphi$. L'algorithme conclut que ϕ est fausse.

Les algorithmes de *model checking* peuvent être résolus en temps $\mathcal{O}(\mathscr{M} \times \phi)$, soit linéaire en fonction de la taille du modèle et du nombre d'opérateurs de la formule. Cependant, la taille de ce modèle est exponentielle en fonction du nombre de variables. Il s'agit d'un algorithme simple, mais peu optimisé et qui nécessite que le modèle soit représenté explicitement sous forme d'un graphe.

Dans la pratique, les logiciels qui permettent de vérifier des formules CTL utilisent plutôt une autre structure de données appelée *ordered binary decision diagrams* (OBDDs), ou diagramme de décision binaire en français. Cette structure de données utilise des ensembles d'états préférablement aux états individuels. L'idée derrière le *model checking* symbolique est de représenter le système de transitions non pas sous forme d'un graphe, mais sous la forme d'une grande expression logique décrivant toutes les transitions valides entre un état et le suivant dans le modèle. Une façon compacte de représenter une telle formule est avec la structure de données BDD. L'évaluation d'une formule CTL peut être effectuée par une série de manipulations sur ce BDD en fonction de la structure de la formule.

L'idée générale est de passer par-dessus les états redondants d'un arbre binaire. Pour ce faire, il faut regrouper les sous-arbres dont leurs feuilles contiennent les mêmes valeurs en un seul état et ensuite identifier les noeuds contenants des sous-arbres isomorphes. De cette façon, nous obtenons un arbre où les noeuds internes sont étiquetés par des variables et les transitions représentent les évaluations possibles de la variable correspondante. Enfin, les feuilles sont étiquetées par la bonne valeur de la fonction. Pour évaluer une formule, on commence par le noeud du haut. Si la variable identifiée à ce noeud est vraie, on prend l'arc de gauche, mais si elle est fausse on prend l'arc de droite. Lorsqu'on arrive à une feuille, cela correspond à la

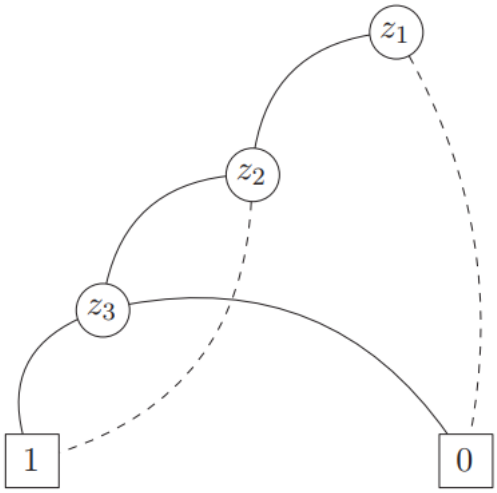


FIGURE 2.9 : Diagramme de décision binaire pour la formule $z_1 \wedge (\neg z_2 \vee z_3)$. Provient du livre *Principles of Model Checking* de Christel Baier et Joost-Pieter Kaoten, réimprimé avec l'autorisation du MIT Press. (Baier & Katoen, 2008)

valeur de vérité de la formule pour les valeurs données aux variables. La figure 2.9 représente un diagramme de décision binaire pour la formule $z_1 \wedge (\neg z_2 \vee z_3)$. Cet algorithme est hors de la portée de ce mémoire, mais on réfère à la section 6.7 de (Baier & Katoen, 2008).

2.3 ALGORITHME LTL

Abordons maintenant le problème de model checking LTL. Le concept de cet l'algorithme est de tenter de trouver un chemin dans $\mathscr{M}$ prouvant $\neg\phi$. S'il n'est pas en mesure de trouver une trace prouvant $\neg\phi$, alors ϕ est valide. L'algorithme de model checking LTL se divise en plusieurs étapes :

— Construire un automate équivalent à la formule LTL $\neg\phi$ qu'on notera $A_{\neg\phi}$,

— Combiner le modèle $\mathscr{M}$ de départ à l'automate nouvellement créé pour $\neg\phi$. Cette opération est le produit $\mathscr{M} \otimes A_{\neg\phi}$.

— Vérifier s'il existe un chemin dans $\mathscr{M} \otimes A_{\neg\phi}$ satisfaisant les conditions d'acceptation de $A_{\neg\phi}$.

2.3.1 LES AUTOMATES DE BÜCHI

Un automate déterministe consiste en un nombre fini d'états et de transitions. Il évalue des exécutions finies et détermine si elles sont valides. Une exécution est valide si l'exécution termine à l'intérieur d'un état acceptant du modèle.

Toutefois, un automate de Büchi déterministe, ou *deterministic Büchi automaton* en anglais, est aussi un automate déterministe, dont la validation des exécutions diffère. Pour qu'une exécution infinie soit valide par un automate de Büchi déterministe, l'exécution doit passer infiniment souvent par un état acceptant, c'est-à-dire qu'il visitera certainement un état acceptant dans le futur.

Pour leur part, les automates de Büchi non-déterministes, *non-deterministic Büchi automata* (NBA) en anglais, admet plusieurs états suivants possible dans la relation de transition.

On définit l'automate de Büchi par $A = (Q, \sum, \delta, Q_0, F, L)$, où :

— Q est un ensemble d'états,

— $\sum$ est un ensemble de symboles appelé l'alphabet,

— $\delta \subseteq Q \times \Sigma \times Q$ la relation de transition,

— Q_0 est un ensemble d'états initiaux,

— F est l'ensemble des états finaux, et

— $L : Q \to 2^{\Sigma}$ est l'étiquetage selon les lettres de l'alphabet $\sum$.

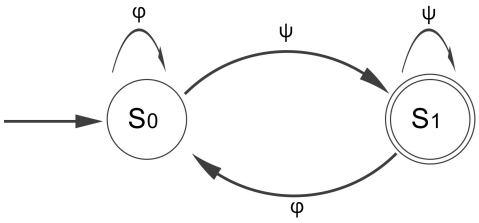


FIGURE 2.10 : Exemple d'un automate de Büchi déterministe.

La figure 2.10 démontre un NBA simple. L'état s_0 est l'état initial et l'état s_1, démontré par le double cercle, est l'état acceptant. De l'état s_0, si la prochaine étape de l'exécution est φ, alors l'automate reste dans le même état. En revanche, recevoir ψ demande au système de transitionner vers s_1. De s_1, ψ lui ordonne de rester dans le même état, mais le système transitionne vers s_0 s'il reçoit φ.

Il était expliqué plus haut qu'une exécution doit visiter infiniment souvent un état final pour que celui soit valide selon le modèle. Par exemple, l'exécution $\{\varphi\},\{\psi\},\{\varphi\},\{\psi\},\{\varphi\},\{\psi\},\{\varphi\},...$ est valide, car il visitera effectivement toujours l'état s_1 dans le futur.

2.3.2 TRANSFORMATION D'UNE FORMULE LTL EN AUTOMATE

Le nouvel automate non déterministe de Büchi (NBA) n'acceptera que les traces satisfaisant $\neg\phi$ et l'automate sera dénoté par le symbole $A_{\neg\phi}$. Cette section est fortement inspirée de la vidéo de B. Srivathsan (Srivathsan, 2015). Générons l'automate non déterministe de Büchi de la formule $\neg\phi = \varphi \mathbf{U} \psi$.

Pour commencer, on liste les sous-formules de $\neg\phi$. Ici, φ, ψ et $\varphi \mathbf{U} \psi$ sont toutes des sous-formules de $\neg\phi$. On souhaite construire une table de vérité pour $\varphi \mathbf{U} \psi$, c'est-à-dire toutes les évaluations de chacune des sous-formules.

φ	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
ψ	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\varphi \mathbf{U} \psi$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

TABLEAU 2.1 : Table de vérité des sous-formules de $\varphi \mathbf{U} \psi$.

La colonne de gauche de la table de vérité correspond aux sous-formules de $\varphi \mathbf{U} \psi$, alors que les autres représentent les états de notre automate. Toutefois, on remarque que certaines vérités sont contradictoires et ne sont pas des états valides : $\varphi \mathbf{U} \psi$ ne peut être vraie si φ et ψ sont fausses comme l'affirme la troisième colonne, $\varphi \mathbf{U} \psi$ ne peut être fausse si φ est fausse et ψ est vraie comme la quatrième colonne et $\varphi \mathbf{U} \psi$ ne peut être fausse si φ et ψ sont vraies. Les états valides sont réunis dans le tableau 2.2.

φ	$\bot$	$\bot$	$\top$	$\top$	$\top$
ψ	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\varphi \mathbf{U} \psi$	$\bot$	$\top$	$\bot$	$\top$	$\top$

TABLEAU 2.2 : Table démontrant les états valides pour $\neg\varphi = \varphi \mathbf{U} \psi$.

Établissons maintenant les transitions possibles entres les états. On a une transition d'un état s vers s' si et seulement si nous avons les conditions suivantes :

— Si $\mathbf{X}\varphi \in$s, alors $\varphi \in s'$

— Si $\neg\mathbf{X}\varphi \in$s, alors $\neg\varphi \in s'$

— Si $(\varphi \mathbf{U} \psi) \in$s et $\psi \notin$s, alors $\varphi \mathbf{U} \psi \in s'$

— Si $\neg(\varphi \mathbf{U} \psi) \in$s et $\varphi \in$s, alors $\neg(\varphi \mathbf{U} \psi) \in s'$

Si nous reprenons notre table de vérité 2.2, la colonne représentant l'état $\top\bot\bot$ ne pourrait avoir de transition vers des états dont $\varphi \mathbf{U} \psi$ est $\top$. Même constat pour la colonne représentant l'état $\top\bot\top$: celui-ci ne pourrait être connecté à d'autres états dont $\varphi \mathbf{U} \psi$ est $\bot$.

2.3.3 CONSTRUCTION DE L'AUTOMATE

On a présentement tous les outils afin de construire l'automate de la formule $\neg\phi$. La figure 2.11 représente les états et transitions possibles de $\varphi \mathbf{U} \psi$ et les états initiaux sont ceux où $\varphi \mathbf{U} \psi$ est vraie, c'est-à-dire s_1, s_2 et s_3.

Les transitions tiennent compte des règles de transitions déclarées précédemment. En effet, l'état s_1 qui correspond à $\top \perp \top$ peut seulement se diriger vers les états s_2 et s_3 où $\varphi \mathbf{U} \psi$ est $\top$. L'état s_5 qui correspond à $\top \perp \perp$ n'a aucune connexion vers des états où $\varphi \mathbf{U} \psi$ est $\top$. Pour le reste, il aucune restriction ne s'applique. Toutes les transitions sont valides.

Il ne reste maintenant qu'à déterminer les états acceptés. Puisque la formule ψ de $\varphi \mathbf{U} \psi$ doit absolument éventuellement devenir vraie, alors seul l'état s_1 n'est pas accepté. On note donc les états s_2 s_3 s_4 et s_5 comme étant les états acceptés.

2.3.4 COMBINAISON DES AUTOMATES

Maintenant qu'on a transformé notre formule LTL en NBA, voyons comment combiner cet automate avec le modèle $\mathscr{M}$. Le résultat de la combinaison est un système de transitions dont les chemins sont à la fois les chemins de l'automate et du modèle. Cette combinaison est aussi appelée un système de transitions de produit $\mathscr{M} \otimes A_{\neg\phi}$. La question est de savoir s'il existe un chemin présent à la fois dans $A_{\neg\phi}$ et $\mathscr{M}$. Si oui, alors la formule LTL de départ n'est pas valide et une trace dans $\mathscr{M}$ existe. En d'autres mots, on veut s'assurer que l'intersection entre le modèle et l'automate soit un ensemble vide ($A_{\neg\phi} \cap \mathscr{M} = \emptyset$). On a ultérieurement défini le modèle $\mathscr{M}$ par $\mathscr{M} = (S, Act, \rightarrow, I, AP, L_1)$ et l'automate de Buchi par $A = (Q, \Sigma, \delta, Q_0, F, L_2)$.

Alors, la définition du produit $\mathscr{M} \otimes A$ est la suivante : $\mathscr{M} \otimes A$ = (S',$\rightarrow$', Q'$_0$ L'), où

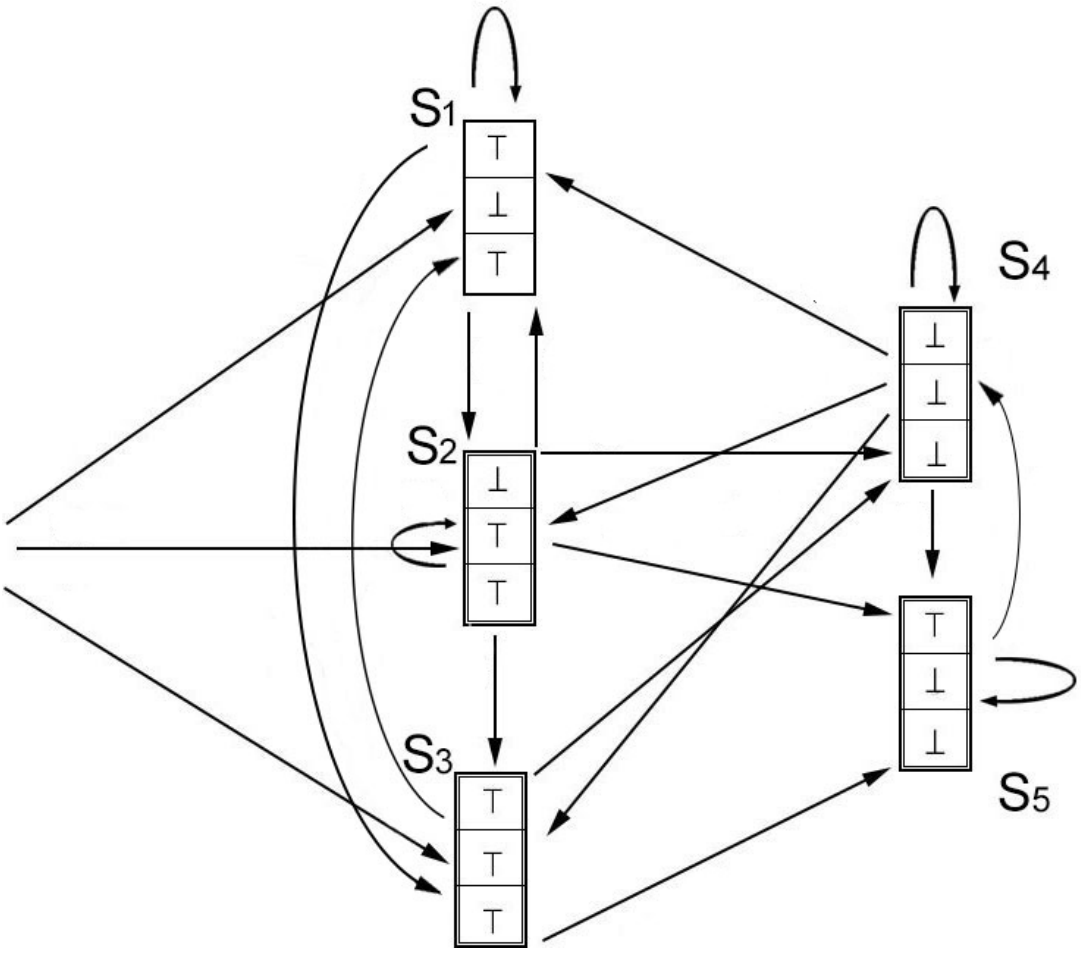


FIGURE 2.11 : Automate final de la formule LTL $\neg\phi = \varphi \,\mathrm{U}\, \psi$.

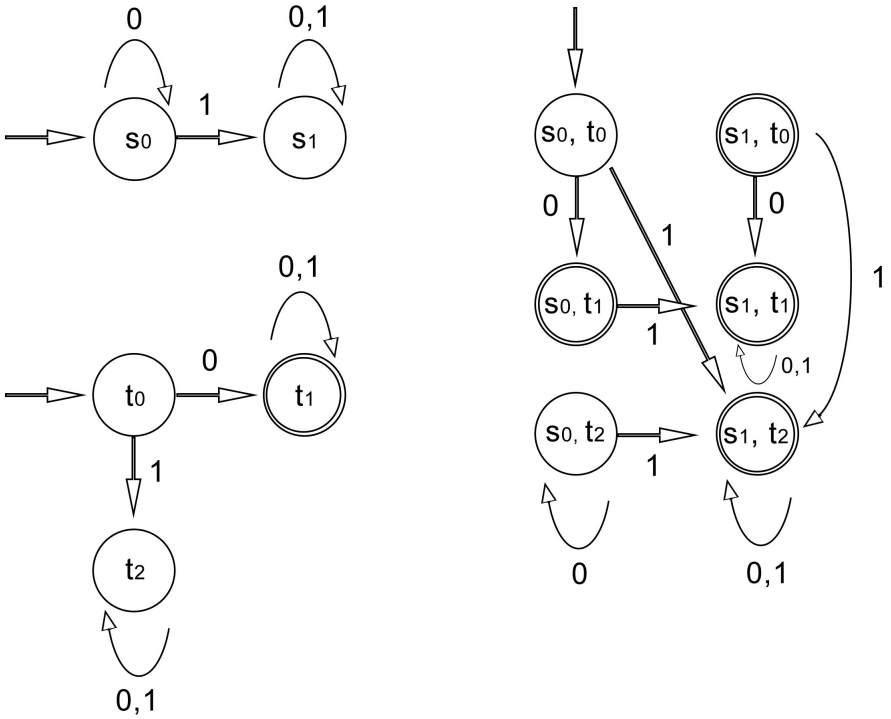


FIGURE 2.12 : À gauche, deux automates. À droite, le produit des deux.

— $S' \subseteq S \times Q$ le produit cartésien des états. Le produit de deux automates aura au maximum |S| × |Q| états.

— $\rightarrow \subseteq S" \times \Sigma \times S'$: définit comme $(s_1, s_2)\sigma(s'_1, s'_2) \in \rightarrow'$ si et seulement si $(s'_1, \sigma, S'_1) \in \rightarrow$ et $(s_2, \sigma, s'_2) \in \delta$

— Q'$_0$ est un ensemble d'états initiaux,

— L' : $(S \times Q) \rightarrow (L_1 \times L_2)$ définie comme $L'(s,q) = (L_1(s), L_2(q))$.

La figure 2.12 montre un exemple de produit entre deux automates. Tout d'abord, on énumère chacune des paires d'états, soit les états $(s_0,t_0),(s_1,t_0),(s_0,t_1),(s_1,t_1),(s_0,t_2)$ et (s_1,t_2). Nous avons un total de 3 × 2 = 6 états. Forcément, l'état initial de l'automate que nous construisons est l'état (s_0,t_0). À partir des états initiaux, recevoir un 0 nous retourne dans l'état s_0 pour le premier automate et dans l'état t_1 pour le deuxième. Il y a donc une transition

de l'état (s_0,t_0) vers (s_0,t_1) marqué par 0. Pareillement pour 1. De s_0, on peut transitionner vers s_1 et de t_0 vers t_2. L'état (s_0,t_0) a une deuxième transition, cette fois-ci vers (s_1,t_2). En répétant ces étapes, nous obtiendrons l'automate de droite de la figure 2.12. Puisqu'il n'y a aucun moyen de se diriger vers les états (s_1,t_0) et (s_0,t_2) à partir de l'état initial, alors ils peuvent être retirés.

2.3.5 RECHERCHE D'UN CHEMIN

Il ne reste qu'à vérifier s'il existe un chemin du système résultant qui satisfait la condition d'acceptation de $A_{\neg\phi}$. On cherche dans l'automate un chemin partant de l'état initial et menant à un état étiqueté avec $\neg\phi$. Si la formule est vraie, un contre-exemple est produit à partir du chemin trouvé. Dans le cas où aucun chemin n'est trouvé, alors la sortie sera $M,s \vDash \phi$ et l'exécution s'arrête ainsi.

Récapitulons. Le problème de model checking se divise toujours par les cinq étapes décrites plus haut : d'abord, lister les sous-formules de $\neg\phi$. Ensuite, à l'aide d'une table de vérité, les évaluer. Les colonnes de cette table définissent les états de l'automate qu'on désire construire, toutefois certaines d'entre-elles ne sont pas valides. On doit donc retirer les colonnes illégales. Lors de la construction de l'automate, on doit suivre les règles de transitions entre les états et ne pas oublier d'afficher les états acceptés. Les états initiaux sont ceux dont $\neg\phi = \top$. Lorsque l'automate est complété, il ne reste qu'à le combiner avec le modèle $\mathscr{M}$ de départ pour ainsi chercher un chemin contenant un état étiqueté par $\neg\phi$.

2.4 LOGICIELS DE *MODEL CHECKING*

Terminons cette section en parlant de quelques logiciels de model checking disponibles dans la littérature. Des détails seront donnés pour NuXMV, SPIN et Java Pathfinder. Il en

existe cependant plusieurs autres, comme FDR2 (Roscoe, 1998), CADP (Garavel, 1989), Alloy (Jackson, 2012) et ProB (Leuschel & Butler, 2003).

NUXMV

NuXMV, anciennement NuSMV (Cimatti *et al.*, 2000), est un vérificateur de modèles populaire dans le domaine. Ce logiciel permet de vérifier les propriétés de modèles finis synchrones en recourant à la logique LTL et CTL vues ultérieurement.

NuXMV utilise son propre langage de spécification appelé SMV (Symbolic Model Verifier). NuXMV prend en entrée un fichier `.smv` comprenant la description des états et des transitions d'un modèle ainsi que les formules CTL et LTL qui doivent être vérifiées sur ce modèle. Il produit ensuite en sortie soit $\top$ ou $\bot$, dépendamment si ces spécifications tiennent selon le modèle. Au cas où la réponse est fausse, un contre-exemple est affiché à l'écran pour expliquer pourquoi celle-ci est fausse lorsque c'est possible. En effet, pour une formule comme $\exists \mathbf{F}\, \phi$, on ne peut pas donner de contre-exemple.

Le fichier smv contient un ou plusieurs modules. Tout comme en C, C#, C++, Java et plusieurs autres langages de programmation, le module MAIN est obligatoire. Il est le coeur du fichier. Dans un module, il est évidemment possible de déclarer et d'initialiser des variables. Pour ce faire, le module doit faire état d'une section nommée `VAR`, exprès pour celles-ci. Les types de variables supportées sont booléens, réels, entiers, énumérations, vecteurs de bits et tableaux. Dans un module, on verra qu'il est possible de référer à un autre module en créant une variable du type du module. Cette technique a été bien utile pour notre projet de recherche afin d'accéder à des variables internes de certains modules qui représentent les processeurs BeepBeep individuellement.

```
MODULE main
VAR
  request : boolean;
  status : {ready,busy};
ASSIGN
  init(status) := ready;
  next(status) := case
                    request : busy;
                    1 : {ready,busy};
                  esac;
LTLSPEC
  G(request -> F status=busy)
```

FIGURE 2.13 : Exemple d'un programme à deux variables. (Huth & Ryan, 2004)

C'est aussi à l'intérieur de ces modules que l'on écrit les transitions du modèle en dessous de l'identifiant ASSIGN. On les modélise à l'aide des mots clés `INIT`, `NEXT` et `TRANS`.

Enfin, le fichier peut se terminer avec une ou plusieurs spécifications. Pour déclarer une formule LTL à vérifier, il suffit de la précéder avec l'identifiant `LTLSPEC`. En revanche, pour une formule CTL, l'identifiant `SPEC` doit être utilisé.

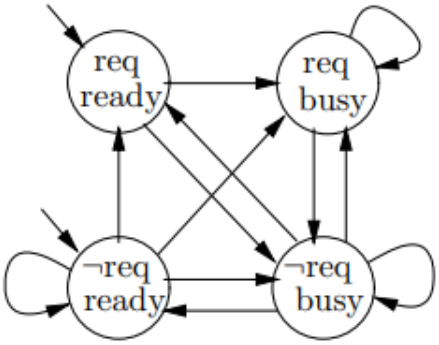


FIGURE 2.14 : Modélisation du programme précédent. (Huth & Ryan, 2004)

La figure 2.13 montre une modélisation d'un exemple simple d'un problème à deux variables. Sous la section `VAR`, deux variables y sont déclarées. La variable `request`, booléenne, et `status`, qui est une énumération à deux valeurs possibles, soit `ready` et `busy`. Sous la section `ASSIGN`, seul `status` est initialisée. C'est donc dire que ce modèle est non déterministe et la variable `request` peut débuter dans n'importe quelles de ses valeurs possibles. Enfin, les transitions de la variable `status` sont les suivantes : si `request` est égale à $\top$, alors `status` est *busy*. Sinon, le programme choisi aléatoirement entre *ready* et *busy*. La figure 2.14 représente cet exemple par un système de transition. Le fichier se termine avec une formule LTL qui vérifie que la variable `status` devient éventuellement *busy* lorsque `request` est égale à $\top$, et ce, à chaque fois.

En ce qui concerne l'identifiant `TRANS`, la relation de transition est spécifiée comme une formule propositionnelle en termes de variables d'état actuelles et suivantes. L'exemple 2.4.1 affirme que si les états sont `ready` et `request`, alors le prochain état est certainement `busy`. L'aspect non-déterministe de la transition est caché par le fait que `request` peut autant être vraie ou fausse.

Exemple 2.4.1. `TRANS` (state = `ready` & `request`) -> next(`state` = `busy`)

Pour terminer, voici quelques commandes utiles donnant de l'information du modèle. Ces commandes sont utilisées dans l'invite de commandes :

— `print_reachable_states` : Affiche le nombre d'états atteignables d'un modèle,

— `check_ctl_spec` : Vérifie la formule CTL sur le modèle,

— `check_ltl_spec` : Vérifie la formule LTL sur le modèle.

SPIN

SPIN (Holzmann, 2004) a été le premier logiciel de *model checking* qui a vu le jour dans les années 1980. Contrairement à NuXMV, les systèmes à vérifier sont décrits en langage Promela. Ce langage supporte la modélisation de modèles distribués asynchrones. Pour la vérification, seule la logique LTL est supportée.

Le logiciel SPIN n'effectue pas directement la vérification de modèle. Il fonctionne en générant un code C qui implémente l'algorithme d'un problème de model checking pour une formule LTL spécifique. Ce processus économise et améliore les performances. L'inconvénient est, pour faire la vérification, le code source doit être compilé à chaque fois que l'on désire exécuter le programme.

JAVA PATHFINDER

Différemment aux deux autres logiciels de vérification, Java Pathfinder (JPF) (Havelund & Pressburger, 2000) sert à vérifier des exécutables Java. Java PathFinder transforme un programme Java en langage Promela, le langage du vérificateur SPIN. Il permet donc aux programmeurs d'annoter leurs programmes avec des assertions et de les vérifier avec SPIN.

JPF utilise un algorithme de recherche brute qui n'a rien à voir avec les algorithmes LTL et CTL décrits plus haut. Dans JPF, une instruction est traitée comme choix et un ensemble de choix est généré lors de la première visite. Les choix correspondent à toutes les valeurs possibles. Un des choix est choisi et le reste est empilé dans une pile. Dès que la fin du programme est atteinte, l'algorithme retourne en arrière au choix le plus récent non exploré et recommence le chemin. L'algorithme se termine lorsque les piles sont vides. Ceci correspond à une exploration en profondeur (depth-first search ou DFS) de l'espace des états du programme

Java. Le noyau de JPF est une machine virtuelle en Java. Similairement à NuXMV, si une erreur est rencontrée, la trace sera affichée.

CHAPITRE III

EVENT STREAM PROCESSING

En entreprise, être témoin d'un processus défectueux n'est pas inhabituel. La plupart du temps, les causes ne sont qu'une simple erreur ou un bogue, mais certains de ces processus défectueux peuvent mener à de graves conséquences. Dans ce genre de situations, les entreprises ont la fâcheuse habitude d'investiguer seulement après que l'événement se soit produit.

L'*event stream processing*, qu'on peut raccourcir par ESP, observe activement et traite des flux d'événements en temps réel. Cette technique permet, par exemple, d'envoyer un signal lorsqu'une valeur critique est reçue comme dans la détection d'événements dans des environnements de communications multimédias (Gao *et al.*, 2010) ou de calculer une grande quantité de données en peu de temps comme dans la prédiction de pannes sur les réseaux et les systèmes (Duenas *et al.*, 2018). L'ESP est composé de trois termes : *Event, stream* et *processing*.

Des *events* (événements) sont des unités de données, qui, lorsqu'elles se suivent, forment le *stream* (flux). Le flux est une chaîne souvent infinie dont les événements pénètrent des unités de calcul et en ressortent transformés. En d'autres mots, c'est l'acte du *processing* (traitement) des données.

Le *stream processing* est donc la pratique d'effectuer des transformations à la chaîne sur une séquence d'événements. Si on dénote par Σ l'ensemble fini d'événements, on dit qu'une trace d'événements est une séquence $\bar{e} = e_0, e_1, \ldots$ où $e_i \in \Sigma \; \forall i$. Lorsqu'une trace d'événements passe à travers une série d'éléments de calcul, on réfère ces éléments à un *pipeline* de stream processing.

Dans ce chapitre, on va présenter les outils les plus populaires appartenant à deux grandes familles de traitement de flux d'événements, soit le *complex event processing* et le *runtime monitoring*.

3.1 COMPLEX EVENT PROCESSING

Le *complex event processing* (CEP) cadre la question du traitement d'un flux d'événements comme un problème de base de données. Un flux d'événements est vu comme une source de données dynamique, sur laquelle des requêtes sont exécutées pour extraire un résultat.

Une caractéristique clé du *complex event processing* est la possibilité de faire des associations entre des événements provenant de multiples sources. Les informations extraites de ces événements peuvent être traitées et conduire à la création de nouveaux événements « complexes » (Hallé, 2016).

Examinons quelques-uns des logiciels existants permettant d'effectuer du CEP. Le logiciel Esper est écrit en Java et peut être ajouté à une application Java. Esper accepte différentes représentations d'événements : Plain-Old Java Object, java.util.Map, tableau d'objets et XML (Mathew, 2014).

Les événements qu'Esper est autorisé à recevoir contiennent de l'information imbriquée. Par exemple, une propriété d'un événement peut elle-même être composée d'un autre événement. Si c'est le cas, Esper utilise le terme *fragment* pour désigner de tels événements (Hallé, 2017). Comme de nombreux logiciels de *stream processing*, Esper utilise un langage fortement inspiré de SQL en empruntant des mots clés comme `SELECT`, `WHERE` et `FROM`. Ces langages sont nommés Event Processing Languages (EPL).

L'exemple suivant est tiré de la documentation d'Esper. On sélectionne un prix total par client parmi les paires d'événements (ServiceOrder suivi d'un événement ProductOrder pour le même ID client dans un intervalle d'une minute) survenant dans les deux dernières heures, où la somme des prix est supérieure à 100, et en utilisant une clause `WHERE` pour filtrer le nom du client.

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
b=ProductOrder(custId = a.custId)
where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

Pour sa part, TelegraphCQ (Chandrasekaran *et al.*, 2003) a été construit pour résoudre le problème du flux continu de données provenant d'environnements en réseau. TelegraphCQ consiste en un ensemble extensible de modules de flux de données composables (ou opérateurs) produisant et consommant des enregistrements de manière analogue aux opérateurs utilisés dans les moteurs de requête de base de données traditionnels. Le traitement des requêtes est effectué en acheminant des tuples de données via des modules de requête. Ces modules sont des bases de données traditionnelles, mais version pipeline. Nous verrons que de nombreux moteurs et langages tiennent pour acquis que les événements ont une structure en tuple.

On peut voir ci-dessous (Hallé, 2016) un fragment de code TelegraphCQ qui correspond à la requête suivante : Pour les cinq journées les plus récentes à partir d'aujourd'hui, sélectionner toutes les actions qui ont clôturées à un prix plus haut que la valeur de Microsoft sur une journée donnée.

```
Select c2.*
FROM ClosingStockPrices as c1, ClosingStockPrices as c2
WHERE c1.stockSymbol = ‘MSFT’ and c2.stockSymbol!= ‘MSFT’ and
c2.closingPrice > c1.closingPrice and c2.timestamp = c1.timestamp
for (t = ST; t < ST + 20 ; t++){
WindowIs(c1, t  4, t);
WindowIs(c2, t  4, t);
```

SASE (Wu *et al.*, 2006) a été développé pour répondre aux besoins d'applications d'identification de radio fréquence. Contrairement à TelegraphCQ, SASE traite plutôt des *pattern queries*, qui décrivent une séquence d'événements qui se produisent dans un ordre temporel et sont corrélés en fonction de leurs attributs. Le code ci-dessous (Hallé, 2016) donne un exemple de pattern query :

```
PATTERN SEQ(TaskStart a,CPU b,TaskFinish c,CPU d)
WHERE a.taskId = c.taskId AND
b.nodeId = a.nodeId AND
d.nodeId = a.nodeId AND
b.value > 95% AND
d.value <= 70% AND
skip_till_any_match(a, b, c, d)
WITHIN 15 seconds
RETURN a, b, c
```

La clause `PATTERN` décrit le modèle d'événements à observer. La clause `WHERE` exprime les conditions des attributs des événements du modèle. Ensuite, le *skip till any match* spécifie

que dans le processus du modèle, les événements ne figurant pas dans le patron seront ignorés jusqu'a ce qu'un événement corrélant avec le modèle soit rencontré. Enfin, la clause `WITHIN` restreint le modèle à l'intérieur d'une période de temps et `RETURN` sélectionne les événements qui seront inclus dans le modèle.

De son côté, Siddhi (Perera *et al.*, 2014) est un moteur de requête pour l'analyse d'événements en temps réel. Il prend en charge la détection d'événements classiques des modèles tels que des filtres, des fenêtres, des jointures ainsi que des modèles d'événements et des fonctionnalités plus avancées telles que les partitions d'événements, l'association de la base de données aux événements, etc.

Siddhi représente les événements à l'aide de tuples. Son architecture se compose de processeurs connectés via des files d'attente. Les événements entrants sont placés dans une file, et les processeurs écoutant ces files traitent ces événements en les plaçant selon les files d'attente de sortie de ce processeur, qui seront ensuite traités par d'autres processeurs ou envoyés aux utilisateurs finaux sous forme de notifications d'événements.

En termes de capacités de requête, Siddhi prend en charge le calcul de fonctions d'agrégation typiques (somme, moyenne, etc.). Il peut également exprimer des modèles séquentiels d'événements, similaires à ceux de SASE, mais en utilisant une syntaxe différente. La requête suivante (Hallé, 2016) en montre un exemple :

```
select f.symbol, p.accountNumber, f.accountNumber
from pattern [every f=FraudWarningEvent2 ->
p=PINChangeEvent2(accountNumber = f.accountNumber)]
```

Cette requête relie deux événements f et p, tels que p doit suivre f et l'attribut accountNumber des deux doivent être identiques. Lorsqu'un tel patron est observé, la requête produit un événement de sortie contenant le symbole et le numéro de compte spécifique à ce modèle.

Il existe beaucoup d'autres engins de CEP ; nous en nommons rapidement quelques-uns dans ce qui suit. Cayuga est un logiciel de CEP pour les flux de données à haut débit (Brenna *et al.*, 2009). Aurora (Carney *et al.*, 2002) définit huit opérations primitives, telles que *window (sliding, latch, tumble, resample)* qui interpole de nouveaux tuples entre les tuples originaux du flux d'entrée. *Filter* filtre les tuples dans un flux pour ceux qui satisfont un prédicat, *map* applique une fonction sur tous les tuples d'un flux, *group by* partitionne les tuples sur de multiples flux et *joins* paire des tuples du flux d'entrée dont la différence dans le temps tombe à l'intérieur d'un intervalle. Enfin, Borealis (Çetintemel *et al.*, 2016) est une version plus récente et multiprocesseurs d'Aurora. Dans Borealis, les unités de traitement sont dotées de lignes de commande spéciales pouvant transporter de l'information permettant de changer le comportement de ces unités durant l'exécution.

3.2 RUNTIME MONITORING

Un domaine apparenté au CEP est celui du runtime monitoring, qui tire ses origines non pas dans les bases de données, mais plutôt dans la communauté des méthodes formelles qui l'a développé comme technique de test. Un moniteur surveille un flux de données selon une spécification d'une propriété que l'on souhaite surveiller. Le flux de données peut provenir du système en temps réel ou d'une file préenregistrée. Le moniteur est donc responsable de donner un verdict à savoir si le flux de données respecte ou non la propriété.

Dans ce qui suit, nous allons présenter brièvement quelques-uns des moniteurs les plus communs dans la littérature et donner des exemples de propriétés que l'on peut exprimer au

moyen du langage qu'ils fournissent. Tout d'abord, LOLA (D'Angelo *et al.*, 2005) est un langage de spécification et un ensemble d'algorithmes pour la surveillance en ligne et hors ligne des systèmes synchrones tels que les circuits et systèmes embarqués. Une spécification LOLA est un ensemble d'équations sur des variables typées. L'exemple suivant (Hallé, 2016) montre une spécification LOLA, résumant la plupart des fonctionnalités du langage.

```
s1 = true
s2 = t3
s3 = t1  (t3  1)
s4 = ((t3)2 + 7) mod 15
s5 = ite(s3; s4; s4 + 1)
s6 = ite(t1;t3  s4; ¬s3)
s7 = t1[+1; false]
s8 = t1[1;true]
s9 = s9[1; 0] + (t3 mod 2)
s10 = t2  (t1  s10[1;true])
```

L'exemple définit dix flux, basés sur trois variables indépendantes t_1, t_2 et t_3. Une expression de flux peut faire intervenir la valeur d'un flux préalablement défini. Les valeurs des flux correspondant à s_3 à s_6 sont obtenus en évaluant leur expression par endroit à chaque position. Le langage fournit une syntaxe du type ite (b ; s_1 ; s_2) qui équivaut à une déclaration *if-then-else*. La valeur retournée dépend si le prédicat du premier opérande retourne la valeur *true*. Le flux correspondant à s_7 est obtenu en prenant à chaque position i la valeur du flux correspondant à t_1 à la position $i+1$, sauf à la dernière position qui prend la valeur par défaut (*false*).

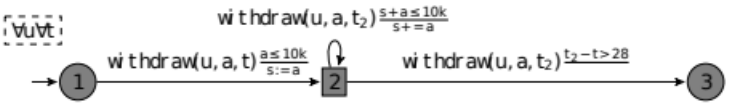


FIGURE 3.1 : Un exemple de *Quantified Event Automata*, utilisé dans MarQ. (Reger *et al.*, 2015)

MarQ (Barringer *et al.*, 2012) est un outil de runtime monitoring qui traite des propriétés paramétriques, dans lesquelles les événements peuvent transporter des données. Un événement paramétrique est une paire formée d'un nom d'événement et d'une liste de valeurs de données et une trace paramétrique est une séquence finie de tels événements. Une propriété paramétrique désigne un ensemble de traces, de la même manière qu'une expression régulière décrit un ensemble de symboles de séquences (Hallé, 2017).

Le *Quantified Event Automata* (QEA) est la notation pour décrire les propriétés paramétriques. La figure 3.1 montre un exemple d'un automate correspondant à la propriété selon laquelle un utilisateur doit retirer moins de 10 000$ à l'intérieur d'une période de 28 jours (Reger *et al.*, 2015).

LARVA (Colombo *et al.*, 2009) utilise comme principal langage de spécification une forme dynamique d'automates communicants avec minuterie et événements, appelés DATEs (dynamic automata with timers and events). Dans ce contexte, les événements peuvent être des *system actions* visibles (telles que les appels de méthode ou la gestion des exceptions), événements minutés, synchronisation de canaux (par laquelle différents automates peuvent se synchroniser) ou une combinaison de ces éléments.

La figure 3.2 montre un exemple de DATE pour une propriété qui monitore de mauvaises connexions se produisant dans un système. Les transitions sont protégées par des conditions sur l'événement d'entrée ; optionnellement, une transition peut mettre à jour des variables internes

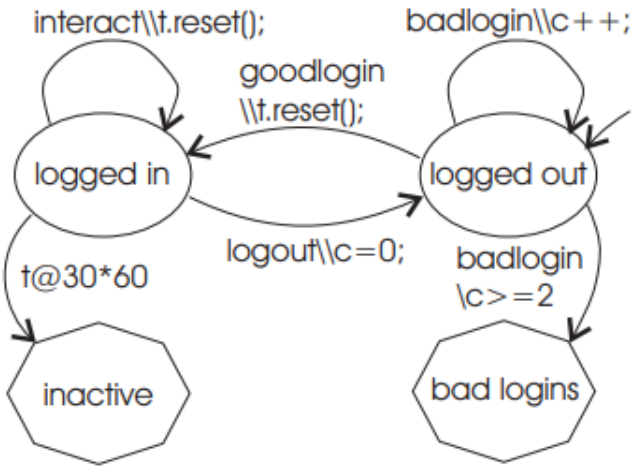


FIGURE 3.2 : Un exemple d'automate, utilisé dans LARVA.
(Colombo *et al.*, 2009) © 2009 IEEE appear

spécifiques à chaque automate. L'intérêt dans les DATEs est la possibilité de déterminer des événements d'expiration comme le démontre le «t@30*60» dans la transition la plus à gauche, déterminant que cette transition est à prendre automatiquement, si aucune autre transition n'a été déclenchée au cours des 30 dernières minutes (Hallé, 2017).

Les spécifications de MonPoly (Basin *et al.*, 2015) sont appelées des formules de *Metric First-Order Temporal Logic* (MFOTL) qui est une extension de LTL avec prédicats et quantificateurs du premier ordre. Dans MonPoly, chaque événement est vu comme une mini base de données qui peut être interrogée au moyen de prédicats. Par exemple, une expression comme *withdraw*(u ; a) est vraie si l'événement courant représente un retrait effectué par l'utilisateur *u* pour un montant *a*.

En plus des connecteurs booléens, les assertions de base peuvent comprendre également des opérateurs temporels et les quantificateurs $\forall$ et $\exists$. Ceci permet de construire des propriétés riches impliquant à la fois des modèles séquentiels et des fonctions d'agrégation. Par exemple, la figure 3.3 vérifie que pour tous les utilisateurs, le nombre de retraits maximaux dans les 31

$$\Box \forall u : \forall c : [\mathsf{CNT}_j\ v; p; \kappa : [\mathsf{AVG}_a\ a; \tau.\blacklozenge_{[0;31)} \\ \text{withdraw}(u; a) \wedge \text{ts}(\tau)](v; u) \wedge \\ \blacklozenge_{[0;31)} \text{withdraw}(u; p) \wedge \text{ts}(\kappa) \wedge 2 \cdot \vee \prec p](c; u) \rightarrow c \preceq 5$$

FIGURE 3.3 : Un exemple de formule, utilisée dans MonPoly. (Hallé, 2017)

derniers jours n'excède pas le seuil de cinq, où ce retrait maximum est une valeur au moins deux fois supérieure à la moyenne des 31 derniers jours (Hallé, 2017).

Enfin, JavaMOP s'appuie sur les concepts de la programmation orientée aspect (AOP) (Kiczales, 1996) pour récupérer les événements pertinents de l'exécution d'un système. En JavaMOP, un événement correspond à un *pointcut*, tel qu'un appel de fonction, un retour de fonction, la création d'un objet, l'affectation d'un champ, etc. Le comportement du système est exprimé en propriétés (comme le nom de l'événement) ; facultativement, une transition peut mettre également à jour les variables internes propres à chaque automate tels que les compteurs.

La figure 3.4 montre une spécification utilisant la logique temporelle. Trois événements atomiques (*hasnexttrue, hasnextfalse* et *before*) sont créés à partir de coupures correspondant à des appels de méthode sur des objets itérateurs. Une spécification LTL stipule alors que tous les événements prochains doivent être précédés par *hasnexttrue*. La section @violation peut contenir du code Java qui doit être exécuté lorsque la spécification est violée. L'ensemble de la spécification est inclus dans une déclaration qui est paramétrée par *i* (Hallé, 2017).

```
HasNext(Iterator i) {

  event hasnexttrue after(Iterator i) returning(boolean b) :
            call(* Iterator.hasNext())
            && target(i) && condition(b) { }
  event hasnextfalse after(Iterator i) returning(boolean b) :
            call(* Iterator.hasNext())
            && target(i) && condition(!b) { }
  event next before(Iterator i) :
            call(* Iterator.next())
            && target(i) { }

  ltl: [](next => (*) hasnexttrue)

  @violation { System.out.println("ltl␣violated!");}
}
```

FIGURE 3.4 : Une spécification JavaMOP. (Hallé, 2017)

3.3 BEEPBEEP 3

BeepBeep 3 est un engin de *stream processing* développé au LIF et écrit en Java (Hallé, 2018). Il vise à concilier le CEP et le runtime monitoring dans un langage de requête uniforme, cohérent et expressif pour les flux d'événements. BeepBeep cherche à fournir un petit nombre d'unités élémentaires de calcul, appelées processeurs, reposant sur la composition pour obtenir des calculs complexes.

Les processeurs BeepBeep sont représentés visuellement par des boîtes ayant un ou plusieurs tuyaux d'entrée et de sortie permettant d'introduire les événements dans le pipeline puis de les récupérer. Les couleurs des tuyaux sont utilisés pour désigner le type d'événements qui les traversent. Selon la convention de la documentation BeepBeep 3, un tuyau bleu-vert représente un flux de nombres ; un tube gris contient un flux de valeurs booléennes, etc.

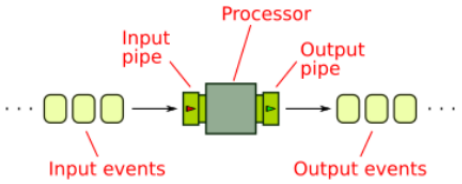


FIGURE 3.5 : Représentation graphique générique d'un processeur prenant un flux d'événement en entrée et produisant un flux de sortie. (Hallé, 2018)

Le nombres de tuyaux d'entrée et de sortie est appelé l'arité (d'entrée et de sortie) du processeur. Par exemple, la figure 3.5 représente un processeur d'arité d'entrée et de sortie 1. Les événements entrent du côté gauche et en ressortent à l'autre bout.

On dénote par $\bar{e} \preceq \bar{e}'$ la séquence $\bar{e}$ étant le préfixe de $\bar{e}'$. Si Σ_1, ..., Σ_n sont n alphabets d'événements, un tuple $\vec{e} = (\bar{e}_1, ..., \bar{e}_n)$ où $\bar{e}_i \in \Sigma_i^*$ pour $i \in [1, n]$ est appelé un vecteur du flux d'événements. Pour deux vecteurs de flux d'événements $\vec{e} = (\bar{e}_1, ..., \bar{e}_n)$ et $\vec{e}' = (\bar{e}'_1, ..., \bar{e}'_n)$, on étend la notation $\vec{e} \preceq \vec{e}'$ si $\bar{e}_i \preceq \bar{e}'_i$ pour tout i ∈ [1, n].

Un processeur peut être défini comme une fonction $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \to (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$. Les valeurs m et n sont l'arité de l'entrée et de la sortie, représentées pas la notation $m : n$. Une fonction est type de relation qui accepte des arguments et qui produit une valeur de retour.

Cependant, pour qu'une telle fonction représente fidèlement le fonctionnement d'un processeur π, nous devons ajouter la condition que pour deux vecteurs d'entrée $\vec{e}_1$ et $\vec{e}_2$, nous rajoutons la condition $\vec{e}_1 \preceq \vec{e}_2$. Celle-ci implique que $\pi(\vec{e}_1) \preceq \pi(\vec{e}_2)$. La structure des processeurs de BeepBeep 3 impose donc qu'aucun événement ne doit se perdre du début jusqu'à la fin d'un pipeline. Si un processeur produit un événement en sortie après avoir consommé un événement en entrée, alors la seule action possible est d'ajouter de nouveaux

événements à la fin de la séquence qui a déjà été produite. De ce fait, on dénote par $\vec{v}$ un vecteur d'événements et par $\pi(\vec{v})$ le vecteur produit en sortie par un processeur.

PRINCIPAUX PROCESSEURS BEEPBEEP 3

Avec une telle notation, donnons les définitions formelles des processeurs essentiels à BeepBeep 3. Nous prenons le temps de bien les définir, car ceux-ci ont été modélisés dans le cadre de notre travail.

CountDecimate est un processeur d'arité 1:1. Pour chaque succession de k événement, le processeur en conserve un et est défini par :

$$\pi(\vec{v} \cdot (e)) \triangleq \begin{cases} \pi(\vec{v}) \cdot e & \text{si } |\vec{v}| \equiv 0 \pmod{k} \\ \pi(\vec{v}) & \text{sinon} \end{cases}$$

Ce processeur, comme ceux qui suivront, satisfait la condition $\pi(\vec{\varepsilon}) \triangleq \vec{\varepsilon}$, où $\vec{\varepsilon}$ signifie un vecteur de flux vide. Autrement dit, chacun des processeurs doivent recevoir au moins une entrée pour produire une sortie.

Le processeur *Trim* enlève du flux d'événements les premiers k événements. Il peut être utilisé pour défausser le premier événement correspondant d'un calcul. Il est défini par :

$$\pi(\vec{v} \cdot (e)) \triangleq \begin{cases} \varepsilon & \text{si } |\vec{v}| \leq k \\ \pi(\vec{v}) \cdot e & \text{sinon} \end{cases}$$

Filter est un processeur $\pi : (\Sigma \times \{\top, \bot\})^* \to \Sigma^*$ qui écarte des événements basés sur un flux de valeurs booléennes. Les événements à la position n du premier flux sont envoyés à la sortie si et seulement si à la même position dans le deuxième flux la valeur booléenne est vraie. *Filter* est défini par :

$$\pi(\vec{v} \cdot (e,b)) \triangleq \begin{cases} \pi(\vec{v}) \cdot e & \text{si } b = \top \\ \pi(\vec{v}) & \text{si } b = \bot \end{cases}$$

Certains processeurs peuvent être utilisés à des fins d'agrégation sur un ensemble d'événements. Le processeur *Cumulate* est conçu pour accumuler des valeurs successives d'une fonction binaire. Soit une fonction $f : \Sigma^2 \to \Sigma$ et une valeur initiale $e_0 \in \Sigma$, le processeur est défini récursivement par :

$$\pi(e) \triangleq f(e_0, e)$$
$$\pi(\bar{e} \cdot e) = \pi(\bar{e}) \cdot f(\pi(\bar{e})[-1], e)$$

Où $\pi(\bar{e})[-1]$ définit le dernier événement produit par π sur le flux d'entrée $\bar{e}$. Cette construction générique peut représenter différents types de calculs selon la fonction utilisée. Par exemple, si f est une addition et e_0 la valeur d'entrée, π produit un flux de sortie ou le i^{ème} événement est la somme de tous les événements d'entrée jusqu'au i^{ème}. Si f est une conjonction booléenne et $e_0 = \top$, π produit un flux en sortie ou le i^{ème} événement est la conjonction de tous les événements jusqu'au i^{ème}.

Le processeur *Window* est certainement le plus complexe de tous, puisqu'il est paramétrisé par un autre processeur BeepBeep $\pi' : \Sigma^* \to \Sigma'^*$. Ce processeur est utilisé pour évaluer un sous-flux de k événements successifs de l'entrée globale du flux :

$$\pi(\bar{e} \cdot e) \triangleq \begin{cases} \varepsilon & \text{si } |\bar{e}| < k-1 \\ \pi(\bar{e}) \cdot \pi'(\bar{e}[-(k-1)..] \cdot e)[-1] & \text{sinon} \end{cases}$$

La notation $\bar{e}[-(k-1)..]$ dénote le suffixe de $\bar{e}$ commençant de l'événement à l'index $k-1$.

Les processeurs définis jusqu'ici ne produisent qu'un seul flux en sortie. Cependant, le processeur *Fork* permet de dupliquer le flux qu'il reçoit en entrée en plusieurs flux en sortie. Il est d'arité 1:n et est défini par : $\pi(\vec{v}) \triangleq \langle \vec{v}, \ldots, \vec{v} \rangle$.

Le processeur *ApplyFunction* prend n'importe quelle fonction de la forme $f : \Sigma_1 \times \cdots \times \Sigma_m \to \Sigma'_1 \times \cdots \times \Sigma'_n$ et en fait un processeur $\pi : (\Sigma_1 \times \cdots \times \Sigma_m)^* \to (\Sigma'_1 \times \cdots \times \Sigma'_n)^*$. Le processeur est défini comme

$$\pi(\vec{v} \cdot (e_1, \ldots, e_m)) \triangleq \pi(\vec{v}) \cdot f(e_1, \ldots, e_m)$$

Enfin, le processeur *Group* permet d'encapsuler un pipeline en entier et le donne en argument à un autre processeur. Tous les processeurs définis ci-haut sont représentés visuellement dans la figure 3.6.

Le pipeline à trois processeurs et trois tuyaux de l'image 3.7 représente un exemple trivial de ce que BeepBeep 3 peut accomplir. Ce pipeline est composé du processeur *Filter*,

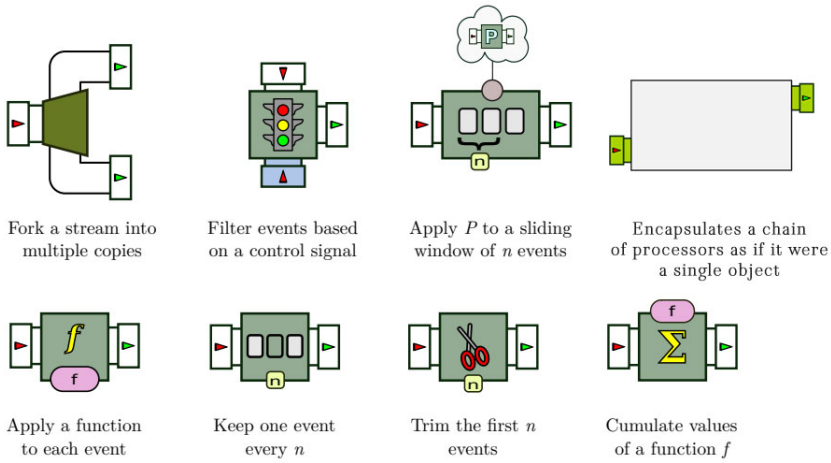


FIGURE 3.6 : Processeurs de BeepBeep 3. Tirés du livre (Hallé, 2018)

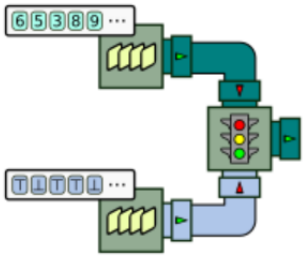


FIGURE 3.7 : Pipeline mettant en vedette le processeur Filter. Les événements 6, 3 et 8 seront produits en sortie par ce processeur. (Hallé, 2018)

dont les entrées sont reliées à des processeurs appelés *Queue Source*. Celui du haut produit des valeurs entières et est connecté à la première entrée du *Filter*, tandis que celui du bas produit les valeurs booléennes et est connecté à la deuxième entrée. En sortie, ce pipeline ne sort que les valeurs entières associées aux valeurs $\top$ du flux de booléens.

Examinons maintenant un exemple plus complexe. L'image 3.8 est la représentation d'un pipeline où le dernier processeur calcule en sortie la somme des événements aux positions i et $3i$. Le pipeline est composé de cinq tuyaux et trois processeurs. Nous remarquons tout de suite que l'extrémité du tuyau 1 n'est relié à aucun processeur. C'est donc dire que le flux

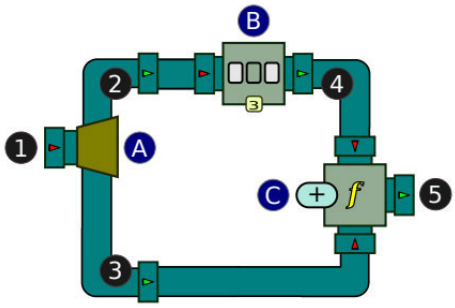


FIGURE 3.8 : Exemple de pipeline calculant la somme à la position *i* et *3i*. (Bédard & Hallé, 2023)

d'événements du pipeline proviendra d'une source externe. Le premier tuyau est l'entrée du processeur *Fork*. Les tuyaux 2 et 3 connectent les sorties de ce processeur aux entrées des processeurs *CountDecimate* et *Add* respectivement. Le tuyau 4 relie la sortie du *CountDecimate* à la première entrée du processeur d'addition, tandis que tuyau 5 représente la sortie globale du pipeline.

Ces processeurs sont élémentaires, mais permettent plusieurs applications : un premier domaine que BeepBeep a permis de faire de la vérification en temps est réel est dans les jeux vidéos. D'abord, *Infinite Mario Bros* est un jeu de plateforme similaire à Mario Bros, mais écrit en Java. Avec l'aide de la logique temporelle, un grand nombre de propriétés a été implémenté afin de savoir le nombre de contraintes que BeepBeep pouvait surveiller lors d'une exécution, telles que la hauteur du saut de Mario, le fait qu'une boule de feu lancé par le joueur doit absolument disparaitre après que celle-ci ait touché un ennemi et que si Mario saute sur la tête d'un joueur allié, celui ne doit pas mourir.

Pingus est un jeu de puzzle libre de droit dont l'objectif est de guider les *pingus* toujours droit devant vers la sortie. Pour réussir, le joueur clique sur les personnages afin de changer leurs états, comme déployer un parachute lorsqu'ils tombent, ou encore de les ordonner de creuser à travers les obstacles. Le processeur *ParseXml* de BeepBeep a été utilisé, puisque ce

jeu utilise l'envoi de messages XML. Un exemple de propriété qui a été vérifiée était l'analyse des états des *pingus* lorsque ceux-ci tombent de haut afin de s'assurer qu'ils meurent lorsqu'ils atterrissent (Boussaha, 2019).

BeepBeep a aussi été utilisé dans le domaine du traitement de signal. Ce scénario touche au concept d'intelligence ambiante, qui est une approche multidisciplinaire qui consiste à valoriser un environnement (bâtiment, voiture, etc.) avec la technologie (capteurs infrarouges, tapis de pression, etc.), afin de construire un système qui prend des décisions basées sur des informations en temps réel et des données historiques au profit des utilisateurs dans cet environnement. Le principal défi de l'intelligence ambiante est la reconnaissance d'activité, qui consiste à capter des données, les filtrer et les transformer en informations pouvant être associées aux activités de la vie quotidienne d'un patient. BeepBeep permet la modélisation de pipeline rapidement pour découvrir les pics sur le signal électrique (Hallé *et al.*, 2016).

Pour terminer ce chapitre, mentionnons qu'il existe des extensions à BeepBeep qui fournissent des processeurs spécifiques à différentes situations. Ces extensions, aussi appelées palettes, sont des bibliothèques supplémentaires qui définissent de nouveaux processeurs ou fonctions à utiliser avec les éléments de base de BeepBeep. Par exemple, il est possible de représenter des machines à états finis, lorsqu'un flux est déterminé par des événements représentant une séquence d'actions et qu'il serait intéressant de vérifier si ces actions suivent un modèle prédéfini et qui stipule dans quel ordre les actions dans ce flux peuvent être observées pour être considérées comme valides.

Une seconde extension permet l'ajout de la logique du premier ordre et de logique temporelle. Cette palette rajoute des processeurs exprimant les opérateurs LTL que nous avons vu. Enfin, d'autres palettes pour la mise en réseau, la représentation graphique et bien d'autres sont disponibles.

CHAPITRE IV
MODÉLISATION DE PIPELINE EN NUXMV

Le problème de ce projet consiste à faire du *model checking* sur des pipelines de processeurs BeepBeep. Les principaux défis sont de transformer un pipeline en structure de Kripke et de trouver une méthode afin que cette transformation soit entièrement automatisée, et ce, peu importe le pipeline à adapter.

Le logiciel de *model checking* que nous avons choisi s'appelle NuXMV. Le travail consiste donc à générer un fichier `.smv` interprétable par NuXMV qui est la traduction d'un pipeline BeepBeep en structure de Kripke. Nous avons antérieurement défini ce type de fichier comme étant la description des états et des transitions d'un modèle ainsi que les spécifications qui lui sont attachées.

On trouvera dans ce chapitre la retranscription des processeurs BeepBeep en modules NuXMV et la modélisation des connexions entre ces modules. De plus, on expliquera un des plus gros défi qu'on devait relever, soit la modélisation des files d'attente. On terminera ce chapitre sur les ajouts apportés dans l'implémentation Java de BeepBeep 3.

4.1 DESCRIPTION DE LA SOLUTION

D'abord, NuXMV permet à un utilisateur de définir des modules qui peuvent être vus comme une forme primitive d'objet. Un module possède des paramètres d'entrée et de sortie, des variables internes et sa propre relation de transition définie en fonction de ces variables. Cette structure nous permet de faire correspondre chaque type de processeur BeepBeep à un module NuXMV.

Nous attachons les modules des processeurs par des pipelines internes. Ceux-ci sont représentés par un triplet de variables. Une première définie le contenu dans le pipeline. Ce contenu peut être soit un booléen ou une valeur entière dépendamment du processeur qui le reçoit. Une deuxième est une valeur booléenne qui détermine s'il y a bel et bien une valeur dans le tuyau et une troisième étant aussi un booléen qui détermine si la valeur actuelle est le dernier événement du flux.

La communication des événements entre les processeurs est modélisée par la présence de ces variables en paramètre lors de l'initialisation des modules. Forcément, le nombre de paramètres dépend de l'arité d'un processeur. Par exemple, un processeur d'arité 1:1 est constitué d'un triplet de variables représentant le tuyau d'entrée et d'un autre triplet représentant celui de sortie ; un processeur d'arité 2:3 est composé de deux triplets d'entrée et de trois de sortie, etc. Si l'arité est déterminée par $m : n$, le nombre de paramètres d'un module sera toujours déterminé par $(3 \times m + 3 \times n) + 1$. Le dernier paramètre est un *reset* utilisé par le processeur *Window* afin qu'il puisse réinitialiser ces modules en mémoire.

Pour indiquer au modèle que deux modules sont connectés par un tuyau, alors un triplet de variables de sortie d'un module ainsi qu'un triplet de variables d'entrée d'un autre doivent concorder. Ainsi, voyons maintenant en profondeur la traduction des processeurs BeepBeep en structure de Kripke.

4.1.1 TRADUCTION

Voyons comment on a retranscrit les processeurs en états et transitions. La modélisation de chaque processeur est fastidieuse et plutôt technique. Ce mémoire n'ira pas dans les détails de la modélisation de chacun des processeurs, mais on en expose quelques-uns pour en illustrer l'idée.

La figure 4.1 montre une modélisation d'un processeur *CountDecimate*. Tout d'abord, un module NuXMV est toujours déclaré avec le mot-clé `MODULE`. Suit ensuite son nom : le nom des modules affirme toujours clairement quel processeur il représente. De plus, dans certain cas, un chiffre complète l'appellation du nom. Dans le cas de notre exemple, le module se nomme `CountDecimate2`. Le 2 est un indicateur que ce module garde un événement tous les deux événements. D'autres processeurs comme *Trim* et *Fork* auront aussi un indicateur : dans le cas de *Trim*, le chiffre indique qu'il supprime un nombre fixe d'événements à partir du début d'un flux et celui du *Fork* indique le nombre de divisions du flux d'origine.

Un pipeline construit de plusieurs processeurs *CountDecimate* pourrait réutiliser le même module si leur fonctionnement est identique. Toutefois, un *CountDecimate* gardant un événement tous les trois événements devra avoir son propre module. En effet, celui-ci ne peut réutiliser un module `CountDecimate2` et un module nommé `CountDecimate3` sera ajouté au modèle.

Les variables du module sont initialisées sous le mot-clé `VAR`. Pour le *CountDecimate*, une seule variable est nécessaire et elle agit comme un compteur. Sous le mot-clé `ASSIGN`, on y retrouve toutes les transitions du module. D'abord, le compteur est initialisé à 0 dès que le module reçoit son tout premier événement. Cette information est comprise à l'intérieur de la déclaration `init(recvd)`. Les transitions de cette variable se retrouve dans la déclaration `next(recvd)`. Si dans le prochain état on reçoit du contenu, alors la valeur de ce compteur équivaut à la formule (recvd + 1) mod 2. Dans le cas d'un `CountDecimate3`, la formule à satisfaire est (recvd + 1) mod 3.

```
MODULE CountDecimate2(inc_1, inb_1, inl_1, ouc_1, oub_1, oul_1, reset)
  VAR
  recvd : 0..1;
  ASSIGN
  init(recvd) := case
    inb_1 : 0;
    TRUE  : 1;
  esac;
  next(recvd) := case
    next(reset) : 1; -- interval-1
    next(inb_1) : (recvd + 1) mod 2;
    TRUE        : recvd;
  esac;
  ...
```

FIGURE 4.1 : Code de départ de la modélisation du processeur *CountDecimate*.

4.1.2 CONNEXION DES PROCESSEURS

Un pipeline interne est ce qui connecte deux modules ensemble et permet à des valeurs de voyager à l'intérieur des pipelines. Ils sont représentés par un triplet de variables utilisées dans l'instanciation des processeurs. Les variables sont nommées selon la nomenclature de deux lettres suivie d'un chiffre : *pc_0, pb_0* et *pl_0*. La lettre *p* signifie *pipeline*. Les suffixes définissent respectivement l'événement contenu (c) à l'intérieur du tuyau, un booléen (b) à savoir s'il y a bel et bien un événement dans le tuyau et un deuxième booléen (l) qui est un témoin déterminant si l'événement est le dernier (*last*) de la séquence. Cette variable est généralement ignorée dans notre modélisation, sauf dans le cas du *QueueSource* qui envoie le signal lorsque son dernier événement est lancé. Lorsqu'un processeur reçoit l'information qu'il traite le dernier, il envoie également l'information à son prochain et ainsi de suite. Compte tenu de cette modélisation, il est important de mentionner qu'un seul événement peut être contenu dans un tuyau. Enfin, le chiffre est l'ID du tuyau dans le pipeline afin de le différencier. Pour sa part, *pc* peut être soit de type booléen ou entier dépendamment du processeur. Si cette variable est un booléen, alors on déclare simplement qu'elle est de type booléenne. Toutefois,

```
  MODULE main
VAR
pc_1: 0..N;
pb_1: boolean;
pl_1: boolean;
prst: boolean;
...
pi_1: Fork_2(pc_1, pb_1, pl_1, pc_2, pb_2, pl_2, pc_3, pb_3, pl_3, reset);
pi_2: Decimate_3(pc2, pb_2, pl_2, pc_4, pb_4, pl_4, reset);
pi_3: ApplyFuction(pc_4, pb_4, pl_4, pc_3, pb_3, pl_3, pc_5, pb_5, pl_5, reset);
```

FIGURE 4.2 : Démonstration du module main de la figure 3.8.

si elle est de type entier, alors la documentation de NuXMV exige qu'on la déclare en donnant la portée des valeurs possibles. Pour le projet, il était donc important de bien cibler nos valeurs, puisque la taille des variables influence le nombre d'états du modèle et donc les performances. Nous y reviendrons plus en détail dans le chapitre 5 qui présente des résultats expérimentaux.

Les entrées et sorties globales du modèle seront représentées par les variables *ic_0, ib_0* et *il_0* pour les entrées et *oc_0, ob_0* et *ol_0* pour les sorties. Les règles de nomenclature et de déclaration des variables sont les mêmes que ci-haut. La figure 4.2 montre le module main de la construction du pipeline de la figure 3.8.

Par la suite, nous avons bénéficié du concept de paramètres pour nos modules. Un tuyau d'entrée ou de sortie d'un processeur sera représenté par un triplet de paramètres du module. Les paramètres des connecteurs d'entrée des processeurs sont toujours représentés par le triplet de variables *inc, inb* et *inl* à l'intérieur des modules. Les deux premières lettres (in) signifient *input*. Dans le cas d'un processeur d'arité 1:1, il n'y aura qu'un seul triplet de paramètres d'entrée dans son modèle. Toutefois, pour un modèle d'un processeur ayant deux entrées, celles-ci seront représentées par les paramètres *inc_1, inb_1, inl_1, inc_2, inb_2* et *inl_2*. Le concept est le même pour les paramètres des connecteurs de sortie. Toutefois, la nomenclature sera du type *ouc_1, oub_1* et *oul_1*. Les lettres *ou* signifient *output*.

```
MODULE CountDecimate2(inc_1, inb_1, inl_1, ouc_1, oub_1, oul_1, reset)
  VAR
  recvd : 0..1;
  ASSIGN
  init(recvd) := case
    inb_1 : 0;
    TRUE  : 1;
  esac;
  next(recvd) := case
    next(reset) : 1; -- interval-1
    next(inb_1) : (recvd + 1) mod 2;
    TRUE        : recvd;
  esac;
  init(oub_1) := inb_1;
  init(ouc_1) := inc_1;
  init(oul_1) := inl_1;
  next(oub_1) := inb_1 & (next(recvd) = 0 | (next(reset) & recvd = 1));
  next(ouc_1) := case
    next(oub_1) : next(inc_1);
    TRUE        : 0;
  esac;
  next(oul_1) := next(inl_1);
```

FIGURE 4.3 : Suite de la modélisation du processeur *CountDecimate*.

Avec ces notions, terminons les explications de la modélisation du processeur *CountDecimate* avec la figure 4.3. Les paramètres du module correspondent directement aux variables lors de sa déclaration. Les transitions des paramètres sont affirmées sous le mot-clé `ASSIGN`. On indique d'abord que les paramètres *oub*, *ouc* et *oul* sont identiques à leur valeur correspondante d'entrée, puisque le *CountDecimate* retourne toujours la toute première valeur. Pour les prochaines étapes, le paramètre *ouc* est dépendant de *oub* et ce paramètre est vrai lorsque le compteur est réinitialisé à 0. Lorsque *oub* est vrai, alors *ouc* retourne la valeur de *inc*.

4.1.3 LES EXCEPTIONS

La majorité des modules suivent le même format : déclarer les variables internes, initialiser leurs états et décrire leurs transitions. Par contre, deux processeurs font exceptions : *GroupProcessor* et *Window*.

Le premier des deux est relativement simple. Le `GroupProcessor` permet d'encapsuler un pipeline en entier et de le donner en argument à un autre processeur. C'est donc dire qu'à l'intérieur de ce module, nous y retrouverons des variables de pipelines internes et globaux, ainsi que des déclarations de modules.

Le deuxième processeur est le `Window`. Ce processeur est paramétrisé par un autre processeur qui évalue un sous-flux de *k* événements successifs du flux d'entré global. Le comportement habituel de ce processeur dans BeepBeep 3 consiste en un premier événement en sortie lorsque les événements 0 à *k - 1* sur une fenêtre de largeur *k* sont traités. Le deuxième événement est produit lorsque les événements 1 à *k* sur une fenêtre de largeur *k* sont traités et ainsi de suite. La solution que nous proposons est différente et fait aussi usage du booléen *reset* pour le processeur en paramètre.

Le module du processeur `Window` instancie *k*-fois le même module en paramètre. Ceux-ci ont la nomenclature `proc1`, `proc2`, ..., `prock` et leurs paramètres se nomment `w_ci_c`, `w_ci_b`, `w_ci_l`, `w_co_c`, `w_co_b`. `w_co_l` et `w_creset`. Tous les modules internes successifs reçoivent le même flux d'événement, à l'exception d'un nombre croissant d'événements coupé à partir du début. En d'autres mots, le premier processeur reçoit tous les événements, le deuxième tous les événements sauf un, le troisième tous les événements sauf deux, etc. Lorsqu'un processeur a reçu *k* événements, ce qui est calculé par le processeur paramétrisé est produit en sortie par le processeur `Window`. Le booléen `reset` est déclenché et l'instance de ce processeur est remis à son état initial.

4.1.4 DÉFI DES FILES D'ATTENTE

Un des plus gros défis que nous devions surmonter était la modélisation des files d'attente à l'intérieur des processeurs dont le nombre d'entrées est plus grand que 1. Par exemple, le processeur d'addition dans BeepBeep 3, illustré par la figure 4.4, doit contenir au moins un événement dans chacune de ses entrées afin d'être en mesure de produire une somme. Or, tout dépendant du pipeline, cette condition n'est toujours remplie ; dans certains cas, des événements doivent donc être conservés en mémoire par le processeur pour une éventuelle consommation ultérieure, c'est-à-dire dans une transition future du modèle. Pour un processeur ayant deux entrées, voici une énumération des comportements des files d'attente.

— Les deux files sont vides et un événement est poussé dans une des deux entrées. L'événement se place à la première place de la file de l'entrée respective.

— Une file est vide et la deuxième contient des événements. Un événement est poussé dans l'entrée qui en contient déjà. Le nouvel événement se place à la suite dans la file d'attente.

— Une file contient des événements et l'autre est vide. Un événement est poussé dans l'entrée ayant la file vide. Ces événement sont consommés et les files sont vides.

— Une file contient des événements et l'autre est vide. Deux événements sont poussés en même temps, un dans chaque entrée. L'événement dans l'entrée avec la file vide est consommé avec le premier élément de la seconde file. Le nouvel événement se place dans la première place dans la file.

— Les deux files sont vides et deux événements sont poussés, un dans chacune des entrées. Les deux événements sont consommés immédiatement.

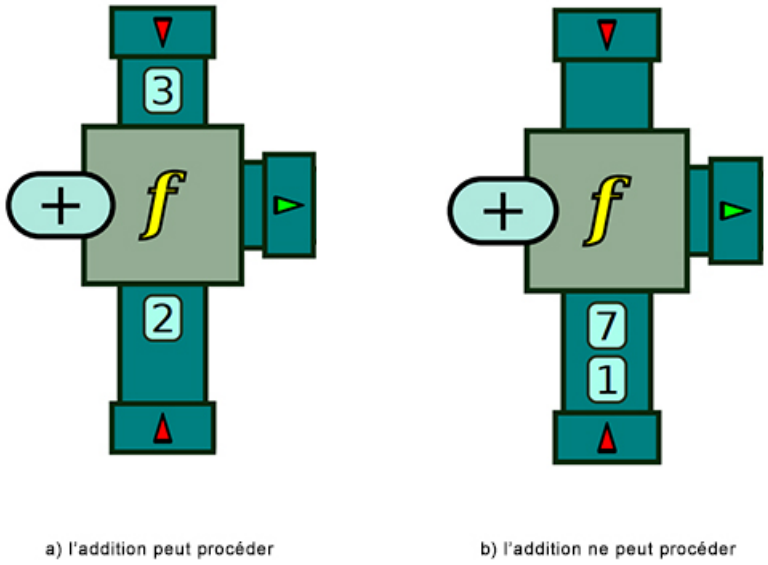


FIGURE 4.4 : Pour qu'une addition produise une valeur, les deux entrées doivent être pourvues d'au moins un événement.

Les files d'attente dans les modules de processeurs ayant un nombre d'entrées plus grand que 1 sont représentées par un tableau de contenu *qc* ainsi qu'un second tableau de booléens *qb*. Suivant la même nomenclature que nous avons vue, les noms des tableaux seront suivis par un chiffre. Donc, les files d'attente du processeur d'addition seront modélisées par *qc_1, qb_1* et *qc_2, qb_2*. Le nombre d'éléments pouvant être stocké à l'intérieur de ceux-ci est déterminé par le deuxième paramètre à l'appel du constructeur de l'objet `SmvCrawler`. Dans le cas où nous décidons d'avoir trois éléments par file d'attente, il est possible de les cibler avec *qc_1[0], qc_1[1], qc_1[2], qb_1[0], qb_1[1], qb_1[2], qc_2[0], qc_2[1], qc_2[2], qb_2[0], qb_2[1], qb_2[2].*

Sachant que nous avons quatre tableaux pour un processeur à deux entrées et que celles-ci comportent trois éléments, c'est donc dire que nous devons modéliser 4×3 variables seulement pour les files d'attente de ce processeur. La variation de la longueur des files d'attente impactera grandement la rapidité des calculs.

Ce choix de modélisation des files d'attente sous la forme d'un tableau rencontre une limite si le modèle d'un processeur ne consomme pas assez rapidement les éléments dans les files d'attente pour ce qu'il reçoit. En effet, si un élément est poussé dans une entrée alors que la file respective est pleine, l'élément sera perdu. À remarquer que cette situation est également possible avec un pipeline BeepBeep réel. En effet, même si un processeur possède des files d'attente en principe illimitées, en réalité il est possible qu'une file d'attente épuise la mémoire de la machine virtuelle Java et provoque l'arrêt du programme. Il ne s'agit donc pas d'une limite spécifique au modèle NuXMV.

Enfin, le comportement des files d'attente est le même pour tous les modèles de processeurs en nécessitant. Le comportement le plus simple est l'ajout d'un événement dans une file alors que les deux sont vides. L'élément se placera à l'indice 0 du tableau. À l'étape suivante, si un élément s'ajoute à la deuxième entrée, alors celui-ci se conjuguera avec l'élément en file dans la première entrée. La modélisation doit permettre le déplacement des éléments dans les files. Par exemple, si le premier élément du premier tableau se fait consommer avec l'élément qui vient de rentrer dans la deuxième sortie, alors l'élément à la position *qc_1[1]* doit se déplacer à la position 0 et l'élément à la position *qc_1[2]* doit se déplacer à la position 1.

4.1.5 IMPLÉMENTATION JAVA

On a vu le principe par lequel chaque processeur peut devenir un module et comment ces modules peuvent être connectés. On cherche maintenant à rendre la conversion d'un pipeline construit dans un programme Java totalement automatique dans un fichier NuXMV qui lui correspond. Cela est possible via des modifications au logiciel BeepBeep qui sont décrites dans ce qui suit.

Les processeurs sur lesquels nous avons travaillés implémentent une interface appelée `SmvPrintable`. À l'intérieur de cette interface, on y retrouve la fonction `printSmv`. Celle-ci écrit le module du processeur en question dans le fichier `smv`. Cette fonction prend en paramètre le flux d'impression, l'ensemble des modules présentement écrit dans le fichier afin d'éviter les répétitions, la taille des files d'attente et la taille du domaine. La fonction vérifie d'abord si le module n'a pas déjà été écrit auparavant. Si le module existe déjà, alors le processeur ne s'écrira pas. Elle se termine en retournant un objet de type `SmvModule` pour populer l'ensemble des modules écrits dans le fichier. Un type `SmvModule` détient certaine information à propos du module, comme le nom et une liste des variables internes avec leur type et longeur respective (dans le cas d'un tableau).

Afin d'identifier les processeurs présents dans un pipeline, nous avons été en mesure de réutiliser un objet BeepBeep 3 nommé `PipeCrawler` et sa fonction `Crawl`. Auparavant, cet objet n'était utilisé que pour la génération du processeur *Group Processeur*. Ces fonctionnalités sont à l'intérieur d'une nouvelle classe appelée `SmvCrawler`.

La fonction `Crawl`, à partir d'un processeur de départ, visite l'entièreté de la chaîne de processeurs. Pour chaque processeur, la fonction vérifie si le processeur dans lequel elle se trouve est instance de `smvPrintable`. Si oui, nous appelons la fonction `printSmv` du processeur. Lorsque `Crawl` a terminé de visiter les processeurs du pipeline, il ne reste qu'à écrire le module `main` avec les variables de connexions avec l'aide des méthodes `printPipe()`, `inferPipeType()`, `printConnections()`, `fetchInputConnections()` et `fetchOutputConnections()`.

Comme mentionné, une connexion est constituée de trois valeurs *c*, *b* et *l*. La fonction `PrintPipe()` se charge d'imprimer dans le fichier `.smv` les variables associées à une

connexion spécifique. Cette fonction appelle `InferPipeType()` afin de déterminer le type de la variable contenu (booléenne ou entière).

Avec l'aide des fonctions `fetchInputConnections()` et `fetchOutputConnections()`, la fonction `PrintConnection()` écrit dans le fichier toutes les connexions. La fonction analyse d'abord quel type de connexion il s'agit (globale ou interne) pour écrire dans le fichier les paramètres correspondants. Enfin, `FetchGlobalInputs()` et `fetchGlobalOutputs()` examinent les entrées et sorties des processeurs et détermine si elle globales. Dans BeepBeep 3, si une entrée est `null`, alors celle-ci est une entrée globale. Complémentairement, si une sortie est `null`, alors elle est une sortie globale.

L'utilisation de la nouvelle extension NuXMV de BeepBeep 3 n'a rien de compliqué. En fait, comme nous pouvons le constater dans la figure 4.5, les quatre dernières lignes de code ont été rajoutées à un fichier existant afin de générer le fichier `.smv`.

Suite à la création du pipeline, la première étape consiste à créer le fichier d'écriture. Suit ensuite l'appel au constructeur de la classe `SmvCrawler`. Les paramètres sont les suivants : le `PrintStream`, afin de permettre aux processeurs d'écrire leur module à l'intérieur du fichier, la taille des files d'attente ainsi que la valeur maximale des événements. Ceci nous permettra plus tard de bien cerner le domaine de nos variables. Si le pipeline ne reçoit que des booléens, cette valeur n'impactera pas la construction du fichier. On appelle finalement la fonction `crawl` avec un processeur de départ. Dans cet exemple, la source du pipeline est le processeur *QueueSource*. Ainsi, cette fonction visitera chacun des processeurs.

Pour finir, voici la liste de tous les processeurs implémentant l'interface `SmvPrintable`. Au final, c'est près d'une trentaine de processeurs BeepBeep que nous avons été en mesure de modéliser en états et transitions : Doubler, Group, ApplyFunction, Cumulate, IfThenElse,

```
    QueueSource source = new QueueSource().setEvents(1, 2, 3, 4, 5, 6);
    Fork fork1 = new Fork(2);
    Connector.connect(source, 0, fork1, 0);

    OutputStream SmvFile = new FileOutputStream("Generation.smv");
    PrintStream printStream = new PrintStream(SmvFile);

    SmvCrawler smvCrawler = new SmvCrawler(printStream, 3, 6);
    smvCrawler.crawl(source);

}
```

FIGURE 4.5 : Exemple d'utilisation de la nouvelle extension NuXMV de BeepBeep 3.

TurnInto, CountDecimate, Filter, Fork, KeepLast, Passthrough, QueueSource, Trim, Window, Booleans, Equals, AbsoluteValue, Addition, IsEven, IsGreaterOrEqual, IsGreaterThan, IsLessOrEqual, LessThan, Minimum, Maximum, Multiplication, Signum et Substraction.

4.2 EXEMPLES D'APPLICATION

Le processus de modélisation des pipelines BeepBeep en structure de Kripke est complet et fonctionnel. Maintenant doté du potentiel d'effectuer une vérification de modèle sur les pipelines BeepBeep, plusieurs applications deviennent possibles.

Dans cette section, on reprend les concepts de cohérence et complétude tels que vu au chapitre 1 et les applique sur des pipelines BeepBeep, on introduit le concept de moniteur dans un moniteur, effectue de la vérification sur les files d'attente, puis compare des pipelines entre eux afin de savoir s'ils produisent le même flux de sortie pour un même flux d'entrée.

4.2.1 PROPRIÉTÉS D'UNE EXÉCUTION

L'étape de vérification de propriétés consiste à construire des formules LTL et CTL valides afin que NuXMV puisse les interpréter. Pour le moment, ces spécifications sont écrites manuellement à la fin du fichier `smv`. Tel que décrit dans le manuel de l'utilisateur de NuXMV, les spécifications LTL et CTL se situent sous les mots-clé `LTLSPEC` ou `CTLSPEC` selon leur type. La figure 2.13 montre un exemple d'utilisation d'une propriété LTL.

Un premier bénéfice de la vérification sur un pipeline de *stream pocessing* est la vérification sur les sorties globales. On peut maintenant prouver la garantie qu'un pipeline produira toujours le résultat attendu pour tous les flux d'intrants possibles. C'est ce qu'on appelle *Correctness*. *Safety* et *Liveness* sont deux types de *Correctness*. La première est la certitude que rien ne peut arriver d'inopportun et *Liveness* est l'assurance qu'au moins un événement se produira dans le futur. Note : pour ces propriétés, on utilise les termes en anglais même dans les publications françaises.

Prenons l'exemple d'un feu de circulation. Une propriété de type *Safety* s'assure qu'une seule direction à la fois peut être dans l'état «lumière verte», tandis que *Liveness* affirme que toutes les lumières deviendront vertes éventuellement. *Bounded Liveness*, une variation de *Liveness*, déclare qu'un processeur doit produire au moins un événement à tous les k étapes. Avec notre exemple de feux de circulation, *Bounded Liveness* pourrait référer au nombre de secondes qui se sont écoulées entre deux feux verts. En effet, on pourrait avoir la certitude, qu'un feu de circulation ne peut passer plus de soixante secondes sans changer au vert.

4.2.2 MONITEUR DANS UN MONITEUR

Comme on l'a dit, le concept de *Bounded Liveness* est intéressant, mais très couteux. Cependant, l'idée d'un pipeline BeepBeep 3 monitorant une structure de Kripke est bien

plus économique. Ceci pourrait remplacer certaines propriétés LTL ou CTL par de simples chaînes de processeurs. Les pipelines de BeepBeep 3 peuvent prendre en entrée des séquences d'événements où ces événements sont des états d'un autre système comme illustré par la figure 4.6. La boîte du haut représente une structure de Kripke quelconque, définit par trois variables *a, b* et *c*. Ces valeurs sont ensuite envoyées dans un pipeline qui produira la valeur $\perp$ dès que le système détectera une violation de la propriété. Un tel concept faciliterait grandement la vérification du *Bounded Liveness*, mais à ce jour, aucun module NuXMV ne permet de compter le nombre de fois qu'un pipeline ne produit aucun d'événement.

Toutefois, considérons la propriété suivante : à n'importe quel moment, *b* contient le nombre de fois que la variable *a* a été vraie dans les *k* états précédents. La formule LTL ressemblerait alors à $\mathrm{G}(a \leftrightarrow \mathbf{X}(a \leftrightarrow \mathbf{X}(a \leftrightarrow \mathbf{X}b = k \vee \neg a \leftrightarrow \mathbf{X}b = k-1)\cdots))$. Le pipeline correspondant surveillant cette propriété est affiché à la figure 4.7. En plus d'être plus simple à représenter, le pipeline est beaucoup plus modulaire que la propriété. En effet, il ne suffirait que de changer la valeur de *k* dans le *Window*, alors qu'une réécriture complète de la formule LTL serait nécessaire.

4.2.3 VÉRIFICATION SUR LES FILES D'ATTENTE

Certains langages de programmation ne permettent pas l'utilisation de files dynamiques afin d'augmenter leur taille lors de l'exécution. Notre implémentation de BeepBeep 3 dans NuXMV peut devenir très utile en ce sens afin qu'elles ne deviennent jamais pleines. On se rappelle que les files d'attente sont représentées par les variables $qb\_i[k]$, où *i* est l'index du processeur et *k* l'emplacement dans la file. Si *Q* est la taille des files spécifiée dans le constructeur de `SmvCrawler`, la propriété CTL pour vérifier cette propriété serait $!(EF(qb\_i[Q-1]))$. Si un modèle satisfait cette expression, alors il est possible de rétrécir la longueur des files

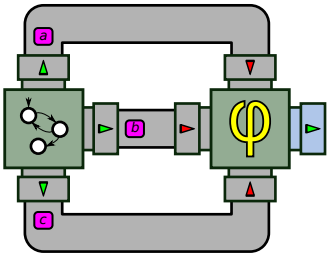


FIGURE 4.6 : Structure de Kripke observée par une chaine de processeurs. (Bédard & Hallé, 2023).

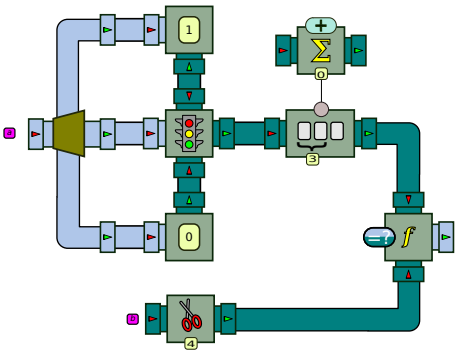


FIGURE 4.7 : Pipeline surveillant une propriété complexe. (Bédard & Hallé, 2021).

d'un emplacement sans danger. Une approche itérative peut alors être employée pour réduire la taille des files d'attente afin de découvrir la taille minimale nécessaire à l'évaluation correcte d'un pipeline sur toute entrée possible.

Il existe cependant des situations dont la taille des files d'attente ne peuvent être bornées. Comme la situation de la figure 3.8 le démontre, la i^{eme} sortie est la somme des événements à la position $3i$ et i. Par conséquent, lorsque la i^{eme} sortie est produite, 3i $-$ 1 événements sont emmagasinés dans la deuxième file du processeur. L'utilisation de la mémoire augmente de façon linéaire, donc aucune limite ne pourra satisfaire la condition.

4.2.4 COMPARAISON ENTRE DEUX PIPELINES

Dans un tout autre ordre d'idée, notre travail offre la possibilité de comparer les chaînes de processeurs entre elles. En effet, on peut maintenant développer des notions de comparaisons entres deux pipelines afin de savoir si elles produisent les mêmes flux en sortie. Cette technique peut être nécessaire si on estime qu'un pipeline peut être simplifié.

Il existe quelques situations où la comparaison de deux pipelines peut être utile. On peut vérifier qu'une simplification appliquée à un pipeline ne perturbe pas son fonctionnement. Par exemple, la figure 4.10 montre un pipeline relativement compliqué formé de 8 processeurs et de 12 tuyaux. En examinant son fonctionnement, on peut découvrir que ce calcul revient en fait à produire une suite de nombres carrés (1, 4, 9, 16, ...). En revanche, la figure 4.11 montre une chaîne de processeurs beaucoup plus simple effectuant le même calcul, composée de 4 processeurs et de 7 canaux. Afin de s'assurer que les deux sont réellement équivalents, ils pourraient être branchés à la place de π et π' sur la figure 4.8.

C'est ce qu'on appelle l'équivalence pas-à-pas : $\pi(\vec{v}) = \pi'(\vec{v})$ pour chaque vecteur de flux $\vec{v}$, π et π' produisent des événements en même temps et le même événement est produit

dans un tel cas. Dans le cas où l'équivalence n'est pas vérifiée, le vérificateur de modèle produit un contre-exemple qui montre une éventuelle entrée violant la condition. Cela peut être utilisé pour aider à identifier la cause de l'écart et résoudre le problème. Si ouc et oub repréprentent la paire de variables correspondant au tuyau de sortie π, et ouc' et oub' représentent la paires de variables correspondant au tuyau de sortie π', alors l'équivalence entre les deux implémentations peut être exprimée selon la formule LTL $\mathbf{G}\,(oub = oub' \wedge (oub' \wedge (oub \rightarrow (ouc = ouc'))))$.

Une condition moins restrictive, appelée équivalence de séquence, affirme que π et π' produisent la même séquence d'événements en supprimant les étapes de calcul où aucun événement n'est produit. Cette condition peut être formulée que pour tout vecteur de flux $\vec{v}$, soit $\pi(\vec{v}) \preceq \pi'(\vec{v})$ ou $\pi'(\vec{v}) \preceq \pi(\vec{v})$. Une telle propriété est habituellement difficile à exprimer directement en logique temporelle : il faut tenir compte du fait que π ou π' ne peut pas produire de résultat à un pas de calcul donné, puis suivre le décalage relatif entre les valeurs de sortie π et π'. Toutefois, cette propriété peut maintenant être facilement représentée par la figure 4.9 qui rajoute simplement un processeur *equal* à la fin du pipeline. Les sorties des processeurs π et π' sont rattachées à ce processeur binaire qui évalue l'égalité entre ses entrées. Grâce au fonctionnement de ce pipeline, les événements produits par π et π' seront mis dans des listes d'attente jusqu'à ce que deux valeurs puissent être comparées sur chaque entrée. Cela a pour effet que les événements aux positions correspondantes dans les deux flux de sortie seront comparés pour l'égalité, quel que soit le nombre d'étapes de calcul nécessaires pour les produire.

En utilisant cette configuration, il est possible de découvrir que les pipelines des figures 4.10 et 4.11 sont équivalents en séquence, mais ne sont pas équivalents au pas à pas. En effet, bien que les deux produisent le même flux de valeurs de sortie, dans le cas du premier pipeline,

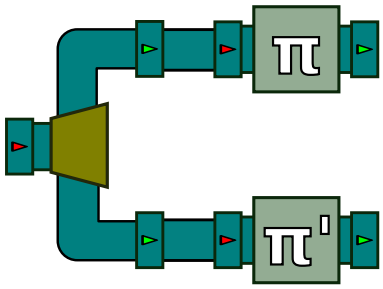


FIGURE 4.8 : L'équivalence pas-à-pas. (Bédard & Hallé, 2023)

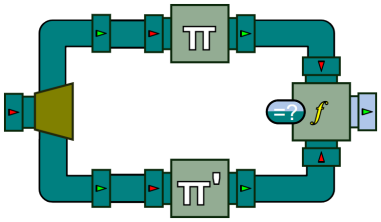


FIGURE 4.9 : L'équivalence de séquence. (Bédard & Hallé, 2023)

il faut pousser deux événements d'entrée pour recevoir le premier événement de sortie. En revanche, le pipeline de la figure 4.12 produit le premier événement de sortie immédiatement après la première poussée. En d'autres termes, le premier flux de sortie est $\varepsilon, 1, 4, 9, \ldots$, tandis que le second est $1, 4, 9, \ldots$

Si l'on remplace simplement la condition paire par impaire dans le pipeline de la figure 4.10, le pipeline simplifié devient celui de la figure 4.11. En effet, le pipeline calcule maintenant la somme de nombres pairs successifs, ce qui peut être accompli avec seulement 3 processeurs et 4 tuyaux. On peut voir que les deux simplifications sont en grande partie sans rapport, en termes de structure, avec le pipeline d'origine, et que même une légère modification de la disposition peut entraîner des simplifications radicalement différentes.

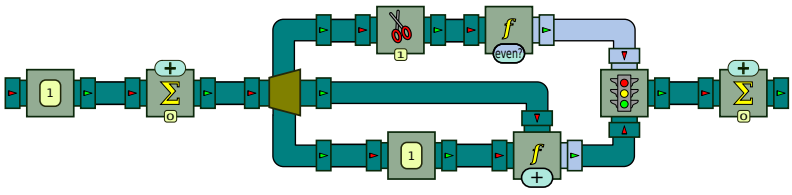


FIGURE 4.10 : Pipeline qui produit une suite de nombres carrés. (Bédard & Hallé, 2023)

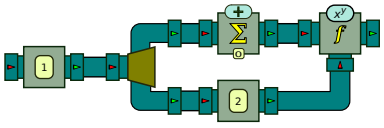


FIGURE 4.11 : Simplification du pipeline original. (Bédard & Hallé, 2023)

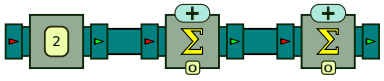


FIGURE 4.12 : Version simplifiée pour les nombres impairs. (Bédard & Hallé, 2023)

CHAPITRE V
ÉVALUATION EXPÉRIMENTALE

Jusqu'à maintenant, on a évacué toute notion de complexité et de performance. Or, on sait que le *model checking* est une opération qui peut être très coûteuse : on se rappelle que les algorithmes de *model checking* peuvent être résolus en temps linéaire en fonction de la taille du modèle, mais que la taille de ce modèle est exponentielle en fonction du nombre de variables. Le but de ce chapitre est d'obtenir un aperçu des ressources consommées lorsqu'on vérifie des propriétés sur de vrais pipelines, au moyen de l'implémentation décrite au chapitre précédent.

5.1 CONFIGURATION EXPÉRIMENTALE

On a conçu nos expériences avec l'assistant LabPal (Halle *et al.*, 2018), une librairie Java open source permettant de réaliser des expériences réutilisables et reproductibles pour des documents de recherche. LabPal facilite également la visualisation des résultats en les interprétant dans des graphes et des tableaux comme ceux de ce chapitre. De façon générale, pour configurer un laboratoire, on doit d'abord créer une expérience. Dans LabPal, les expériences sont toutes des descendantes de la classe `Experiment`. Pour ce projet, cette classe prend trois paramètres ; un `ModelProvider` qui fournit le modèle (le fichier NuXMV que nous avons généré), un `PropertyProvider` qui fournit les propriétés CTL et LTL à évaluer sur le modèle, ainsi que le booléen `with_stats` qui permet à l'expérience de récolter davantage de données sur la taille de l'espace. Notre laboratoire exécute NuXMV en appelant la fonction `runNuSMV`, puis analyse les résultats obtenus du *model checker* avec la fonction `parseStatsResults`. Les données analysées sont le temps de vérification, le verdict calculé par NuXMV sur l'évaluation de la propriété, la mémoire, le nombre total de noeuds du modèle,

le diamètre du graphe, le nombre total d'états et le nombre d'états accessibles. Nos expériences sont disponibles en ligne sur le site de github [1].

Pour cette première itération du projet, on a analysé le temps de complétion de la vérification ainsi que la consommation de la mémoire en faisant varier la taille des files d'attente (Q_b), la largeur du domaine (N) et le paramètre k de certains processeurs, notamment *window*. Lorsqu'on regarde les valeurs testées de Q_b, N et k, cela représente une combinaison de 100 modèles NuXMV. Combinée avec les propriétés à vérifier, l'étendue du nombre de modèles distincts correspond à 2974 problèmes de *model checking*. Les pipelines de processeurs donnent lieu à des structures de Kripke pouvant comporter jusqu'à 19 modules et 77 variables.

Voici une liste des chaînes de processeurs BeepBeep 3 qu'on a testé. La représentation de ces pipelines se retrouve à l'annexe B :

1. *Passthrough* : Une chaîne de processeurs ne contenant qu'un seul processeur *Passthrough*.
2. *Product of window of width k* : Un processeur *Window* multipliant k événements.
3. *Sum of 1s on window* : Calcule la moyenne mobile d'un flux de nombres. La moyenne mobile est un type de moyenne statistique utilisée pour analyser des séries ordonnées de données en supprimant les fluctuations transitoires de façon à en souligner les tendances à plus long terme.
4. *Sum of doubles* : Les nombres sont d'abord multipliés par deux. Ce pipeline additionne ensuite le nouveau nombre avec le total des nombres précédents.
5. *Output if smaller than k* : Retournera en sortie les k premiers événements, puis rien par la suite.

1. https://github.com/alexisBedard/nusmv-beepbeep-lab

6. *Product of 1 and k-th* : Retourne la multiplication entre le premier et le k i^{ème} nombre.

Nom du pipeline	Aucune file pleine	Liveness	Bounded Liveness
Output if smaller than k	469	467	502
Passthrough	443	429	434
Product of 1 and k-th	5408	5892	43221
Product of window of width 3	5263	5231	15983
Sum of 1s on window	1720	1783	5099
Sum of doubles	1689	1657	11150

TABLEAU 5.1 : Temps de vérification en millisecondes des différentes chaînes de processeurs pour les propriétés *aucune file pleine*, *liveness* et *bounded liveness* pour une taille du domaine de 4 et des tailles de files d'attente de 3.

Sur ces chaînes de processeurs, plusieurs propriétés sont considérées : une première vérifie si les files d'attente ne sont jamais pleines ainsi que les propriétés *Liveness* et *Bounded Liveness* telles que décrites dans le chapitre 4. Les paramètres les plus susceptibles d'avoir un impact sur la complexité sont le pipeline considéré, la propriété à évaluer sur ce pipeline, la taille du domaine de ses variables, la taille des files d'attente et dans le cas du processeur *Window* la largeur de la fenêtre. Ce sont ces variables qui sont contrôlées dans nos différentes expériences.

Le but de ces expérimentations était de donner un ordre de grandeur en termes de temps et de consommation de mémoire lors de la vérification de diverses propriétés sur les pipelines, et par conséquent, déterminer dans quelle mesure les paramètres mentionnés ci-dessus ont une influence sur ces mesures. Les expériences ont été exécutées sur un processeur Intel Core i7-8700K 3.70 GHz, 8 GO de mémoire vive, s'exécutant sur Windows 10 et utilisant la version 2.0.0 de NuXMV.

5.2 RÉSULTATS ET DISCUSSION

Les résultats prouvent d'abord qu'il est effectivement possible de vérifier des propriétés sur des pipelines. Le tableau 5.1 montre le temps de vérification des chaînes de processeurs énoncées ci-dessus pour les propriétés *aucune file pleine*, *liveness* et *bounded liveness*. La taille du domaine et les tailles des files d'attente sont fixées à 4 et 3 respectivement. Une analyse de ce tableau nous révèle que le temps de vérification est naturellement plus élevé pour les pipelines contenant des files d'attente, comme *Product of 1 and k-th* et *Sum of doubles* et des processeurs *window*, comme *Product of window of width k* et *Sum of 1s on window*. Une deuxième observation qu'on peut relever est le fait que l'exécution de la propriété *Bounded Liveness* semble effectivement exigeante pour le système comme on l'a initialement énoncé dans le Chapitre 4.2.

Dans l'ensemble, on note qu'il faut au maximum 3 minutes pour faire une vérification d'une propriété. Par contre, on est d'avis qu'une taille d'un domaine de 4 et une taille des files d'attente de 3 n'est pas nécessairement représentatif de la réalité et lorsqu'on tente d'augmenter les valeurs de ces variables, le temps de vérification devient beaucoup trop long et aucun résultat ne peut être extrait des expériences.

5.2.1 IMPACT DES FILES D'ATTENTE

En se basant sur les résultats du pipeline *Product of 1 and k-th* du tableau 5.1, on observe une corrélation entre la taille des files d'attente et l'augmentation des temps de vérification. En fait, on constate que les temps d'exécution augmentent exponentiellement pour les pipelines utilisant des processeurs contenant des files d'attente. À titre indicatif, les tableaux 5.2 et 5.3 fixent la variable N pour ne faire varier que Q_b. Les chiffres montrent que pour une longueur de files d'attente de 4, plus d'une minute est nécessaire pour la complétion de la vérification

des propriétés pour ce pipeline, alors que le temps d'exécution n'était que d'une seconde avec une taille de 2. De plus, il nous a été impossible de vérifier la propriété *Bounded Liveness* sur ces pipelines lorsque la longueur des files dépassait 4. Après une heure d'attente, aucun résultat n'avait été retourné par NuXMV.

Taille des files d'attente	Aucune file pleine	Liveness
1	585	683
2	1004	1052
3	5408	5892
4	78118	91748

TABLEAU 5.2 : Temps en millisecondes du pipeline Product of 1 and k-th lorsque la taille du domaine est 4.

Taille des files d'attente	Aucune file pleine	Liveness
1	451	473
2	736	716
3	1689	1657
4	14961	14957

TABLEAU 5.3 : Temps en millisecondes du pipeline Size for Sum of doubles lorsque la taille du domaine est 4.

On est en mesure de fournir des explications aux variations majeures entre ces temps d'exécution. En effet, elles s'expliquent par l'augmentation du nombre d'états dans le modèle résultant. Pour le pipeline *Product of 1 and k-th* de taille de domaine 4 et de taille de files variant entre 1 et 4, on a un nombre d'états équivalent à 2^{39}, 2^{45}, 2^{51} et 2^{57}. En termes de consommation de mémoire, cela nécessite exactement 9 458 992 octets lorsque Q_b=1, 9 665 232 octets quand Q_b=2, puis 9 952 496 et 10 540 528 octets ensuite.

Fait intéressant, une augmentation des files d'attente n'affecte pas les processeurs paramétrisés à l'intérieur d'un processeur *Window* si ceux-ci sont d'arité $m > 1$. En effet, le

Tailles du domaine et des files d'attente	Nombre d'états	Nombre d'états attaignable	Nombre d'états atteignable en BDD
Q_b=1, N=4	2^{39}	2^{16}	6989
Q_b=2, N=4	2^{45}	2^{21}	9656
Q_b=3, N=4	2^{51}	2^{25}	15800
Q_b=4, N=4	2^{57}	2^{30}	27826

TABLEAU 5.4 : Nombre d'états de la structure de Kripke du pipeline *Product of 1 and t-th* selon les paramètres Q_b et N.

processeur paramétrisé n'utilise jamais ses files d'attente si bien que d'avoir une très grande valeur de Q_b n'impacterait en rien les résultats. Les tableaux 5.5 et 5.6 démontrent la constance des résultats.

Taille des files d'attente	Aucune file pleine	Liveness
1	5284	5304
2	5236	5223
3	5263	5231
4	5353	5344

TABLEAU 5.5 : Temps en millisecondes du pipeline Product of window of width 3 lorsque la taille du domaine est 4.

Taille des files d'attente	Aucune file pleine	Liveness
1	1725	1757
2	1858	1949
3	1720	1783
4	1730	1786

TABLEAU 5.6 : Temps en millisecondes du pipeline Sum of 1s on window lorsque la taille du domaine est 4.

5.2.2 IMPACT DE LA TAILLE DU DOMAINE

Procédons au même exercice, mais en faisant varier la taille du domaine. Si on rassemble les données des tableaux 5.9 et 5.10, on remarque que le temps de calcul n'est pas indifférent à N. Encore une fois, le temps d'attente augmente selon la taille du domaine, mais moins considérablement qu'avec la taille des files d'attente. En effet, faire varier Q_b de 3 à 4 augmentait de 14 fois le temps de vérification. Ici, on ne parle que d'un ratio de 1,6 pour N.

Enfin, contrairement à la conclusion qu'on a tirée selon laquelle la variation de la taille des files d'attente n'affecte pas les processeurs *Window*, cette affirmation ne tient pas pour une variation du domaine. Les tableaux 5.11 et 5.12 montrent les résultats. En effet, un processeur *Window* copie k fois un second processeur et le fait d'augmenter la taille du domaine affecte toutes les copies de leurs variables. Par contre, ce qui taxe le plus les performances des processeurs *Window* est la variation du paramètre k. À titre d'exemple, on a exécuté la vérification des propriétés *aucune file pleine* et *liveness* sur le pipeline *Sum of 1s on window*, alors que les variables Q_b et N sont fixées à 2 et 3 respectivement. Lorsque $k = 2$, le

Largeur de la fenêtre	Aucune file pleine	Liveness
2	299	502
3	930	730
4	7780	8297
5	15803	16262

TABLEAU 5.7 : Temps de vérification en millisecondes pour le pipeline Sum of 1s on window selon la largeur de la fenêtre

Largeur de la fenêtre	Aucune file pleine	Liveness
2	213	305
3	3824	3937
4	41334	41980
5	-	-

TABLEAU 5.8 : Temps de vérification en millisecondes pour le pipeline Product of window of width 3 selon la largeur de la fenêtre

temps d'exécution est égal à 299ms et 502ms. Lorsque $k = 3$, on a des résultats de 930ms et 730ms. Les temps de vérification augmentent à 7780ms et 8297ms quand $k = 4$, puis 15803ms et 16262ms avec $k = 5$. Suivant le même gabarit, mais avec le pipeline *Product of window of width 3*, les résultats sont : 213ms, 305ms, 3824ms, 3937ms, 41334ms et 41980ms. On a effectuer une expérience où $k = 5$ pour ce pipeline, mais impossible d'obtenir un résultat. Les tableaux 5.7 et 5.8 détaillent ces chiffres.

5.2.3 IMPACT DE LA PROPRIÉTÉ

Terminons la discussion avec une dernière suite de tests qui, cette fois-ci, repose sur la comparaison entre des pipelines. Ces comparaisons utilisent les propriétés équivalence pas-à-pas afin de déterminer si les flux en sortie des processeurs π et π' sont identiques et l'équivalence de séquence qui vérifie si les flux sont identiques et qui progressent aux mêmes étapes de calcul. L'expérience qu'on a menée consiste à comparer deux pipelines

différents, mais équivalents. Les résultats dans les tableaux 5.13 et 5.14 montrent que la propriété équivalence de séquence prend beaucoup plus de temps à vérifier que l'équivalence pas-à-pas. Cette comparaison a été menée sur deux paires de pipelines : une première qui simule une fenêtre glissante de largeur 2 et une deuxième de largeur 3, les deux effectuant une addition des entrées reçues.

Taille des files d'attente	Aucune file pleine	Liveness
2	2386	2165
3	3252	3334
4	5408	5892
5	16517	17198

TABLEAU 5.9 : Temps en millisecondes du pipeline Product of 1 and k-th lorsque la taille des files d'attente est 3.

Taille des files d'attente	Aucune file pleine	Liveness
3	705	679
4	1689	1657
5	16333	15459

TABLEAU 5.10 : Temps en millisecondes du pipeline Sum of double lorsque la taille des files d'attente est 3.

Taille des files d'attente	Aucune file pleine	Liveness
2	504	501
3	2399	2449
4	5263	5231
5	22553	22886

TABLEAU 5.11 : Temps en millisecondes du pipeline Product of window of width 3 lorsque la taille des files d'attente est 3.

Taille des files d'attente	Aucune file pleine	Liveness
2	498	512
3	786	798
4	1720	1783
5	4863	4824

TABLEAU 5.12 : Temps en millisecondes du pipeline Sum of 1s on window lorsque la taille des files d'attente est 3.

Pour clore ce chapitre, récapitulons nos résultats tirées de nos expériences. Des tests sur une centaine de modèles NuXMV ont été effectuées et sur ces modèles, on mesure les temps de vérification et la consommation de mémoires. On peut convenir que :

— Le temps de vérification est naturellement plus élevé pour les pipelines contenant des listes d'attente et leur taille font une différence sur celle-ci.

— Le paramètre k impact également le temps de vérification, mais pas à la même échelle que les listes d'attente.

Nom du pipeline	Temps
Window sum of 2 comparison	908
Window sum of 3 comparison	17126

TABLEAU 5.13 : Temps de vérification en millisecondes pour la propriété équivalence pas-à-pas.

Nom du pipeline	Temps
Window sum of 2 comparison	2590
Window sum of 3 comparison	86087

TABLEAU 5.14 : Temps de vérification en millisecondes pour la propriété équivalence de séquence.

— Comparer des pipelines est très demandant pour le système.

CONCLUSION

Dans les dernières années, un bon nombre de logiciels d'*event stream processing* ont été développés, mais aucun d'entre eux ne permet la vérification de propriété sur son exécution. Dans ce mémoire, on a montré qu'il était possible de transformer un pipeline de processeurs BeepBeep 3 en structure de Kripke afin de l'exporter dans un logiciel de *model checking*. Les résultats des expériences du chapitre 5 montrent effectivement la pertinence de l'approche. Au meilleur de nos connaissances, on peut affirmer qu'effectuer de la vérification formelle sur de tels types de calculs est étudiée pour la première fois. Les spécificités de cette traduction restent toutefois dépendantes du système.

Tout au long de ce mémoire, on a introduit des concepts de logique mathématique, de *model checking* et d'*event stream processing* pour ensuite répondre à la question : «est-il possible de représenter un pipeline de stream processing en une structure de Kripke ? »

DISCUSSION ET LIMITATIONS

On a vu dans le chapitre 4 les outils aidant à la génération des fichiers `smv` : la fonction `printSMV` produit les modules `smv`, l'objet `PipeCrawler` visite chacun des processeurs d'un pipeline et `PrintPipe()` se charge d'imprimer les variables associées aux connexions. Ce choix de conception facilite la maintenance de notre extension. En effet, afin de supporter de nouveaux processeurs, ceux-ci doivent seulement étendre de l'interface `SmvCrawler` et posséder la fonction `printSMV`. Aucune autre modification n'est à apporter dans BeepBeep ; le module sera pris en compte dans les prochaines générations.

Pour ce qui est des tailles des listes d'attente, nos expérimentations ont démontré une certaine fragilité. D'abord, NuXMV ne permet pas l'utilisation de tableau dynamique. C'est

donc dire qu'à chaque exécution, on doit fixer une taille aux listes. Par conséquent, il peut être souhaitable de vérifier que la mémoire nécessaire pour évaluer un pipeline ne dépasse jamais le seuil donné.

La modélisation des files d'attente est aussi un enjeu à améliorer et l'exploration d'autres logiciels de *model checking* pourrait être envisagée. On sait que SPIN offre la possibilité d'utiliser des canaux de communication synchrones entre des processeurs. Ceci pourrait être une avenue intéressante qui pourrait éventuellement remplacer notre modélisation construite «à la main». Sans toutefois modéliser tous les processeurs, on pourrait en choisir quelques-uns pour les tester. Il serait en particulier intéressant de tester les processeurs d'arité supérieur à 1 dans SPIN pour ensuite comparer les résultats obtenus avec ceux de NuXMV.

Le choix de NuXMV en général limite l'applicabilité de l'approche pour la vérification de modèles de *stream processing*. Par exemple, il nous a été impossible de modéliser certains processeurs BeepBeep qui utilisent des listes, des ensembles ou des tableaux associatifs. NuXMV permet seulement le calcul sur des données primitives comme les entiers et les booléens. Qui plus est, on a été tenus de ne pas rendre accessible la fonction de division avec le processeur *ApplyFonction*, car à moins de contrôler le flux d'entrée, il était impossible d'éviter l'apparition de nombre fractionnaire. Par contre, une nouvelle version de NuXMV a été rendue publique lors du développement de ce travail. Cette version permet l'utilisation de nombres réels grâce au nouveau type de variable `Real`.

Comme mentionné dans le chapitre 3, plusieurs systèmes d'*event stream processing*, comme Siddhi et Esper, utilisent un langage semblable à SQL. On ne sait pas jusqu'à quel point les processeurs BeepBeep pourraient être modélisés de manière équivalente, car la traduction n'a jamais été examinée. On peut donc présumer que cette approche requerrait une reprise de fond en comble de la modélisation.

TRAVAUX FUTURS

Nous avons mentionné à quelques reprises que les performances laissent à désirer. Afin de les améliorer, instaurer un minimum à la taille du domaine serait une piste de solution. Pour le moment, le constructeur de `SmvCrawler` comporte trois paramètres : le *printStream*, la taille des listes d'attente ainsi que la valeur maximale des événements. Le minimum est automatiquement 0, donc le domaine est de 0 jusqu'au maximum choisit. Cela implique que le domaine est souvent plus grand que ce qu'on a réellement besoin. Un second inconvénient est l'impossibilité de travailler avec des valeurs négatives.

Dans ce travail, on a modélisé une liste limitée de processeurs BeepBeep. Cependant, d'autres processeurs pourraient être simulés. On pense entre autres à la palette LTL de BeepBeep 3. En effet, BeepBeep 3 contient des processeurs évaluant les opérateurs **G**, **F**, **X** et **U**. Une fois ces opérateurs supportés par NuXMV, il serait intéressant de créer un pipeline vérifiant une certaine propriété LTL et, par-dessus, vérifier la même propriété en utilisant NuXMV.

De plus, il nous est impossible pour le moment de savoir avec certitude si notre intégration dans NuXMV est fidèle au code Java de BeepBeep. Pour ce faire, on devrait utiliser Java Pathfinder, afin d'avoir une assurance que par définition, les processeurs BeepBeep 3 implémentés en Java agissent comme le stipulent leur définition formelle. De son côté, la définition des processeurs vers NuXMV est déjà accessible avec les formules CTL et LTL. Ainsi, on aurait l'assurance requise que la définition BeepBeep 3 est équivalente à celle de NuXMV.

Enfin, un nouveau concept à introduire serait celui des canaux avec perte illustrés par l'image 5.1. Ce concept est bien rarement utile dans des environnements synchrones comme

BeepBeep, mais il pourrait simuler des environnements réseau pour montrer les conséquences des pertes de données.

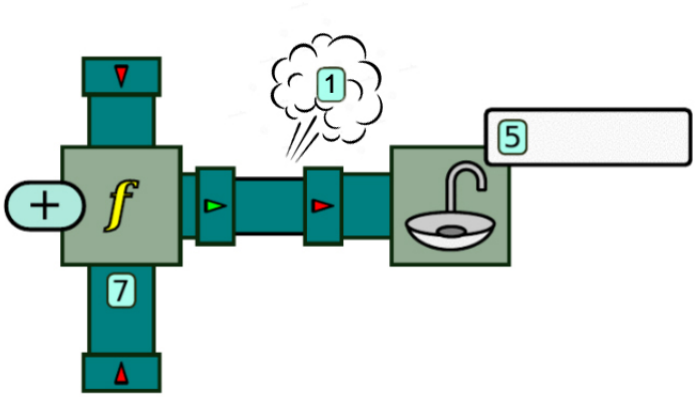


FIGURE 5.1 : Exemple de canal avec perte.

BIBLIOGRAPHIE

Allwood, J., Andersson, G.-G., Andersson, L.-G. & Dahl, O. (1977). *Logic in linguistics*. Cambridge University Press.

Alur, R., Feder, T. & Henzinger, T. A. (1996). The benefits of relaxing punctuality. *Journal of the ACM (JACM)*, *43*(1), 116–146.

Austin, A. & Williams, L. (2011). One technique is not enough : A comparison of vulnerability discovery techniques. Dans *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 97–106. IEEE.

Baier, C. & Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

Barringer, H., Falcone, Y., Havelund, K., Reger, G. & Rydeheard, D. (2012). Quantified event automata : Towards expressive and efficient runtime monitors. Dans *International Symposium on Formal Methods*, pp. 68–84. Springer.

Barzin, M. & Errera, A. (1929). *Sur le principe du tiers exclu*. Imprimerie Stevens Frères.

Basin, D., Klaedtke, F., Marinovic, S. & Zălinescu, E. (2015). Monitoring of temporal first-order properties with aggregations. *Formal methods in system design*, *46*(3), 262–285.

Batra, M. (2013). Formal methods : Benefits, challenges and future direction. *Journal of Global Research in Computer Science*, *4*(5), 21–25.

Bédard, A. & Hallé, S. (2021). Model Checking of Stream Processing Pipelines. Dans C. Combi, J. Eder, & M. Reynolds (Éds.). *28th International Symposium on Temporal Representation and Reasoning (TIME 2021)*, Vol. 206 de *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 5 :1–5 :17., Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.TIME.2021.5

Behrend, A., Dorau, C., Manthey, R. & Schüeller, G. (2008). Incremental view-based analysis of stock market data streams. Dans *Proceedings of the 2008 international symposium on Database engineering & applications*, pp. 269–275.

Berry, A. & Milosevic, Z. (2013). Real-Time Analytics for Legacy Data Streams in Health : Monitoring Health Data Quality. Dans *2013 17th IEEE International Enterprise Distributed Object Computing Conference*, pp. 91–100. doi: 10.1109/EDOC.2013.19

Boussaha, M. R. (2019). *Surveillance des propriétés de sécurité avec BeepBeep*. *Surveillance des propriétés de sécurité avec BeepBeep*.

Brenna, L., Gehrke, J., Hong, M. & Johansen, D. (2009). Distributed event stream processing with non-deterministic finite automata. Dans *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pp. 1–12.

Bédard, A. & Hallé, S. (2023). Formal verification for event stream processing : Model checking of BeepBeep stream processing pipelines. *Information and Computation*, *293*, 105058. doi: https://doi.org/10.1016/j.ic.2023.105058

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. & Tzoumas, K. (2015). Apache flink : Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, *38*(4).

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Tatbul, N., Zdonik, S. & Stonebraker, M. (2002). Monitoring streams—a new class of data management applications. Dans *VLDB'02 : Proceedings of the 28th International Conference on Very Large Databases*, pp. 215–226. Elsevier.

Çetintemel, U., Abadi, D., Ahmad, Y., Balakrishnan, H., Balazinska, M., Cherniack, M., Hwang, J.-H., Madden, S., Maskey, A., Rasin, A. *et al.* (2016). The aurora and borealis stream processing engines. Dans *Data Stream Management* pp. 337–359. Springer.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F. & Shah, M. A. (2003). TelegraphCQ : continuous dataflow processing. Dans *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pp. 668–668.

Cimatti, A., Clarke, E., Giunchiglia, F. & Roveri, M. (2000). NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, *2*, 410–425.

Clarke, J., Edmund, M., Grumberg, O. & Peled, D. A. (1999). Model checking. *MIT Press*, p. 314.

Colombo, C., Pace, G. J. & Schneider, G. (2009). Larva—safer monitoring of real-time java programs (tool paper). Dans *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pp. 33–37. IEEE.

D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H., Mehrotra, S. & Manna, Z. (2005). LOLA : runtime monitoring of synchronous systems. Dans *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pp. 166–174. doi: 10.1109/TIME.2005.26

De Menezes, M. V., do Lago Pereira, S. & de Barros, L. N. (2010). System Design Modification with Actions. Dans A. C. da Rocha Costa, R. M. Vicari, & F. Tonidandel (Éds.). *Advances in Artificial Intelligence – SBIA 2010*, pp. 31–40., Berlin, Heidelberg. Springer Berlin Heidelberg.

Duenas, J. C., Navarro, J. M., Parada G., H. A., Andion, J. & Cuadrado, F. (2018). Applying Event Stream Processing to Network Online Failure Prediction. *IEEE Communications Magazine*, *56*(1), 166–170. doi: 10.1109/MCOM.2018.1601135

Fages, F. (2003). Symbolic model-checking for biochemical systems. Dans *International Conference on Logic Programming*, pp. 102–102. Springer.

Francalanza, A., Aceto, L., Achilleos, A., Attard, D. P., Cassar, I., Della Monica, D. & Ingólfsdóttir, A. (2017). A foundation for runtime monitoring. Dans *International Conference on Runtime Verification*, pp. 8–29. Springer.

Gao, M., Yang, X., Jain, R. & Ooi, B. (2010, 06). Spatio-temporal Event Stream Processing in Multimedia Communication Systems. Vol. 6187, pp. 602–620. doi: 10.1007/978-3-642-13818-8_41

Garavel, H. (1989). *Compilation et vérification de programmes LOTOS*. *Compilation et vérification de programmes LOTOS*.

Hallé, S. (2016). When rv meets cep. Dans *International Conference on Runtime Verifica-*

tion, pp. 68–91. Springer.

Hallé, S. (2017). From complex event processing to simple event processing. *arXiv preprint arXiv :1702.08051*.

Hallé, S., Gaboury, S. & Bouchard, B. (2016). Activity Recognition Through Complex Event Processing : First Findings. Dans *AAAI workshop : artificial intelligence applied to assistive technologies and smart environments*.

Halle, S., Khoury, R. & Awesso, M. (2018). Streamlining the inclusion of computer experiments in a research paper. *Computer*, *51*(11), 78–89.

Hallé, S. (2018). *Event Stream Processing with BeepBeep3*. Repéré à https://liflab.gitbook.io/event-stream-processing-with-beepbeep-3/

Havelund, K. (2008). Runtime verification of C programs. Dans *Testing of Software and Communicating Systems : 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008 8th International Workshop, FATES 2008 Tokyo, Japan, June 10-13, 2008 Proceedings*, pp. 7–22. Springer.

Havelund, K. & Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, *2*, 366–381.

Holzmann, G. J. (2004). *The SPIN model checker : Primer and reference manual*, Vol. 1003. Addison-wesley Reading.

Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A. & Rosu, G. (2014). ROSRV : Runtime verification for robots. Dans *Runtime Verification : 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*, pp. 247–254. Springer.

Huth, M. & Ryan, M. (2004). *Logic in Computer Science : Modelling and reasoning about systems*. Cambridge university press.

Jackson, D. (2012). *Software Abstractions : logic, language, and analysis*. MIT press.

Kacker, R. N., Kuhn, D. R., Lei, Y. & Lawrence, J. F. (2013). Combinatorial testing for software : An adaptation of design of experiments. *Measurement*, *46*(9), 3745–3752.

Karaman, S. (2009). *Optimal planning with temporal logic specifications*. *Optimal planning with temporal logic specifications*.

Kern, L. H., Mirels, H. L. & Hinshaw, V. G. (1983). Scientists' understanding of propositional logic : An experimental investigation. *Social studies of science*, *13*(1), 131–146.

Kiczales, G. (1996). Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, *28*(4es), 154–es.

Klement, K. C. (2004). Propositional Logic. Dans *Internet Encyclopedia of Philosophy*.

Kleppmann, M. (2019). *Designing Data-Intensive Applications*.

Koenig, P., Mangler, J. & Rinderle-Ma, S. (2019). Compliance Monitoring on Process Event Streams from Multiple Sources. Dans *2019 International Conference on Process Mining (ICPM)*, pp. 113–120. doi: 10.1109/ICPM.2019.00026

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, *2*(4), 255–299.

Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I. & Heljanko, K. (2012). Model checking of safety-critical software in the nuclear engineering domain. *Reliability Engineering & System Safety*, *105*, 104–113.

Latvala, T. & Mäkelä, M. (2004). Ltl model checking for modular petri nets. Dans *International Conference on Application and Theory of Petri Nets*, pp. 298–311. Springer.

Leuschel, M. & Butler, M. (2003). ProB : A model checker for B. Dans *FME 2003 : Formal Methods : International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, pp. 855–874. Springer.

Luckham, D. C. (2005). *The power of events – An introduction to complex event processing*

in distributed enterprise systems. ACM.

Maler, O. & Nickovic, D. (2004). Monitoring temporal properties of continuous signals. Dans *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* pp. 152–166. Springer.

Mathew, A. (2014). Benchmarking of complex event processing engine-esper. *Dept. Comput. Sci. Eng., Indian Inst. Technol. Bombay, Maharashtra, India, Tech. Rep. IITB/C-SE/2014/April/61*.

Neumeyer, L., Robbins, B., Nair, A. & Kesari, A. (2010). S4 : Distributed stream computing platform. Dans *2010 IEEE International Conference on Data Mining Workshops*, pp. 170–177. IEEE.

Owens, J. D., Rixner, S., Kapasi, U. J., Mattson, P., Towles, B., Serebrin, B. & Dally, W. J. (2002). Media processing applications on the Imagine stream processor. Dans *Proceedings. IEEE International Conference on Computer Design : VLSI in Computers and Processors*, pp. 295–302. IEEE.

Perera, S., Sriskandarajah, S., Vivekanandalingam, M., Fremantle, P. & Weerawarana, S. (2014). Solving the grand challenge using an opensource CEP engine. Dans *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, pp. 288–293.

Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR)*, *9*(3), 223–252.

Petri, C. A. & Reisig, W. (2008). Petri net. *Scholarpedia*, *3*(4), 6477.

Raghavan, V., Rundensteiner, E., Woycheese, J. & Mukherji, A. (2007). FireStream : Sensor Stream Processing for Monitoring Fire Spread. Dans *2007 IEEE 23rd International Conference on Data Engineering*, pp. 1507–1508. doi: 10.1109/ICDE.2007.369056

Ramchandani, C. (1973). *Analysis of asynchronous concurrent systems by timed petri nets. Analysis of asynchronous concurrent systems by timed petri nets.*

Reger, G., Cruz, H. C. & Rydeheard, D. (2015). MarQ : Monitoring at Runtime with QEA.

Dans C. Baier & C. Tinelli (Éds.). *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 596–610., Berlin, Heidelberg. Springer Berlin Heidelberg.

Roscoe, A. (1998). The theory and practice of concurrency.

Serot, J., Berry, F. & Ahmed, S. (2011). Implementing stream-processing applications on fpgas : A dsl-based approach. Dans *2011 21st International Conference on Field Programmable Logic and Applications*, pp. 130–137. IEEE.

Smullyan, R. (1995). *First-order Logic*. Dover books on advanced mathematics. Dover. Repéré à https://books.google.ca/books?id=kgvhQ-oSZiUC

Srivathsan, B. (2015). *Automaton construction*. Repéré le 2021-10-21, à https://www.youtube.com/watch?v=_0fBHdmpTHA&t=1167s&ab_channel=ModelChecking

Terry, D., Goldberg, D., Nichols, D. & Oki, B. (1992). Continuous queries over append-only databases. *Acm Sigmod Record*, *21*(2), 321–330.

van der Aalst, W. M. (1993). Interval timed coloured Petri nets and their analysis. Dans *International conference on application and theory of petri nets*, pp. 453–472. Springer.

Visser, A. (1981). A propositional logic with explicit fixed points. *Studia Logica*, pp. 155–175.

Wang, J. (1998). Time Petri Nets. Dans *Timed Petri Nets : Theory and Application* (pp. 63–123). Boston, MA: Springer US. doi: 10.1007/978-1-4615-5537-7_4

Wang, J. (2012). *Timed Petri nets : Theory and application*, Vol. 9. Springer Science & Business Media.

Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer*, *23*(9), 8–22.

Wu, E., Diao, Y. & Rizvi, S. (2006). High-performance complex event processing over streams. Dans *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 407–418.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. & Stoica, I. (2013). Discretized streams : Fault-tolerant streaming computation at scale. Dans *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pp. 423–438.

Zhang, W., Freiheit, T. & Yang, H. (2005). Dynamic scheduling in flexible assembly system based on timed Petri nets model. *Robotics and Computer-Integrated Manufacturing*, *21*(6), 550–558.

APPENDICE A

LISTE DE PROCESSEURS BEEPBEEP 3 POUVANT ÊTRE TRADUIT DANS NUXMV

a) applyfunction

b) fonction d'addition

c) fonction de soustraction

d) fonction de multiplication

e) fonction doubler

f) fonction plus grand ou égal que

g) fonction plus grand que

h) fonction plus petit ou égal que

i) fonction plus petit que

j) fonction égal

k) fonciton valeur absolue

l) fonction maximum

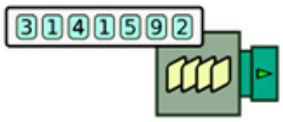
m) fonction minimum

o) fonction Signum

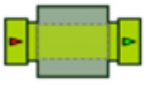
p) fonction and

q) fonction or

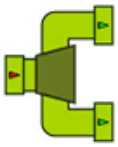
r) fonction not



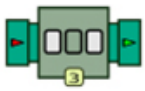
s) processeur queuesource



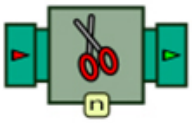
t) processeur passthrough



u) processeur fork

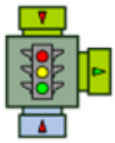


v) processeur count decimate



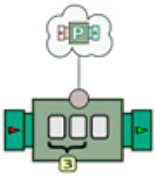
v) processeur trim

w) processeur cumulate



x) processeur filter

y) fonction if then else



z) processeur window

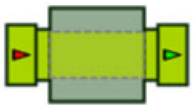
aa) processeur group processor

bb) processeur keep last

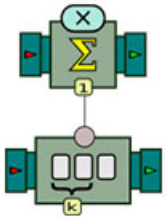
cc) processeur turn into

APPENDICE B

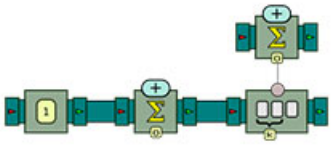
REPRÉSENTATION VISUELLE DES PIPELINES SUR LESQUELLES NOUS AVONS EXÉCUTÉ NOS EXPÉRIENCES



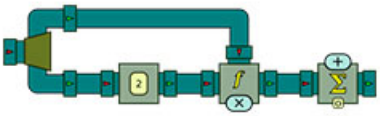
a) Passthrough



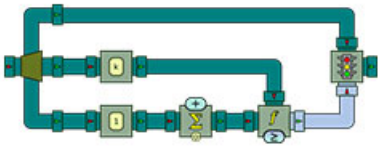
b) Product of window of width k



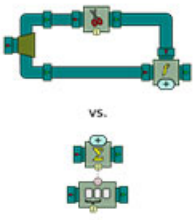
c) Sum of 1s on window



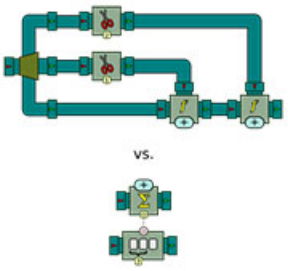
d) Sum of doubles



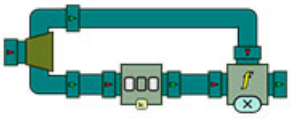
e) Output if smaller than k



f) Window sum of 2 comparison



g) Window sum of 3 comparison



h) Product of 1 and k-th

APPENDICE C

RÉSULTATS EXPÉRIMENTAUX

Domain size	No full queues	Liveness
4	469	467
5	485	490

a) Running time by domain size for output if smaller than k (queues = 3)

Domain size	No full queues	Liveness
2	2386	2165
3	3252	3334
4	5408	5892
5	16517	17198

b) Running time by domain size for product of 1 and k-th (queues = 3)

Domain size	No full queues	Liveness
2	447	450
3	444	454
4	443	429
5	441	441

c) Running time by domain size for Passthrough (queues size= 3)

Domain size	No full queues	Liveness
2	504	501
3	2399	2449
4	5263	5231
5	22553	22886

d) Running time by domain size for product of window of width 3 (queues = 3)

Domain size	No full queues	Liveness
3	705	679
4	1689	1657
5	16333	15459

e) Running time by domain size for Sum of doubles (queues = 3)

Domain size	No full queues	Liveness
2	498	512
3	786	798
4	1720	1783
5	4863	4824

f) Running time by domain size for Sum of 1s on window (queues = 3)

Queue size	No full queues	Liveness
1	436	446
2	423	430
3	443	429
4	445	435

g) Running time by queue size for Passthrough (domain = 4)

Queue size	No full queues	Liveness
1	462	447
2	459	466
3	469	467
4	496	484

h) Running time by queue size for output if smaller than k (domain = 4)

Query	No full queues	Liveness	Bounded liveness
Output if smaller than k	469	467	502
Passthrough	443	429	434
Product of 1 and k-th	5408	5892	43221
Product of window of width 3	5263	5231	15983
Sum of 1s on window	1720	1783	5099
Sum of doubles	1689	1657	11150

i) Running time by processor chain

Queue size	No full queues	Liveness
1	1725	1757
2	1858	1949
3	1720	1783
4	1730	1786

j) Running time by queue size for Sum of 1s on window (domain = 4)

Queue size	No full queues	Liveness
1	585	683
2	1004	1052
3	5408	5892
4	78118	91748

k) Running time by queue size for Product of 1 and k-th (domain = 4)

Queue size	No full queues	Liveness
1	5284	5304
2	5236	5223
3	5263	5231
4	5353	5344

l) Running time by queue size for Product of window of width 3 (domain = 4)

Queue size	No full queues	Liveness
2	465	479

m) Running time by value of k for Output if smaller than k (domain = 3, queues = 2)

Queue size	No full queues	Liveness
2	1346	1345

n) Running time by value of k for product of 1 and k th (domain = 3, queues = 2)

Queue size	No full queues	Liveness
1	451	473
2	736	716
3	1689	1657
4	14961	14957

o) Running time by queue size for Sum of doubles (domain = 4)

Query	Time
Window sum of 2 comparison	2590
Window sum of 3 comparison	86087

p) Running time for sequence equivalence

Queue size	No full queues	Liveness
2	8255	8549

q) Running time by value of k for Sum of 1s on window (domain = 3, queues = 2)

Queue size	No full queues	Liveness
2	256712	261759

r) Running time by value of k for product of window of width 3 (domain = 3, queues = 2)

Query	Time
Window sum of 2 comparison	908
Window sum of 3 comparison	17126

s) Running time for stepwise equivalence