UQAC
Université du Québec
à Chicoutimi

DÉCENTRALISATION DU CONTRÔLE DE CONFORMITÉ DE PROCESSUS OPÉRATIONNELS : APPROCHES PAR BLOCKCHAIN ET SÉQUENCE DE PAIR-ACTIONS

PAR QUENTIN BETTI

THÈSE PRÉSENTÉE À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI EN VERTU D'UN PROTOCLE D'ENTENTE AVEC L'UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS, EN VUE DE L'OBTENTION DU GRADE PHILOSOPHIÆ DOCTOR (PH.D.) EN SCIENCES ET TECHNOLOGIES DE L'INFORMATION

QUÉBEC, CANADA

RÉSUMÉ

La gestion de processus d'affaires (ou BPM, pour Business Process Management) fait l'objet d'un intérêt majeur et croissant dans l'industrie et la recherche académique. La principale explication à ce phénomène réside dans le fait que le BPM améliore grandement les performances et la flexibilité globales de l'exécution de processus opérationnels. Cette amélioration repose en partie sur la possibilité d'automatiser les contrôles de conformité des processus, qui assurent que ces derniers suivent une exécution valide par rapport aux modèles mis en place. En outre, depuis une quinzaine d'années, on a pu observer l'avènement des processus opérationnels centrés sur les artefacts. Dans cette approche, plus intuitive et flexible, l'enchaînement des opérations possibles est exprimé par l'ensemble des contraintes, appelé *cycle de vie*, que les pairs intervenant dans le processus doivent respecter lorsqu'ils interagissent avec les objets physiques ou virtuels manipulés en son sein, appelés *artefacts*. Contrôler la conformité d'un tel processus revient alors à vérifier que les cycles de vie des artefacts sont respectés, et ce à toutes les étapes du processus.

Dans ce domaine, les travaux de recherche et les solutions industrielles actuelles se basent néanmoins principalement sur des architectures centralisées, qui représentent plusieurs inconvénients en termes de sécurité et flexibilité. L'objet de cette thèse est donc de proposer des approches embrassant une décentralisation complète du procédé de vérification de conformité. La première d'entre elles consiste à stocker la totalité des manipulations sous la forme de transactions incluses dans une blockchain Ethereum. En effet, la blockchain permet l'établissement de registres décentralisés, immuables et authentifiés. Plus précisément, les actions sont stockées par l'intermédiaire de *smart contracts*, programmes partagés et exécutés par tous les nœuds du réseau blockchain. Nous appliquons cette architecture à un scénario concret de livraisons de colis dans un contexte de logistique hyperconnectée. Plusieurs propriétés de cycle de vie pertinentes dans ce milieu sont exprimées, implémentées et vérifiées à l'aide de BeepBeep, une bibliothèque de Complex Event Processing (CEP).

Bien que fonctionnelle, l'approche par blockchain expose des inconvénients majeurs, notamment en termes du nombre d'actions qu'il est possible de traiter dans un temps donné, jugé insuffisant dans certains contextes. Nous proposons alors une approche originale où les actions sont stockées directement dans l'artefact manipulé sous la forme d'une *séquence de pair-actions*. Les séquences de pair-actions permettent de construire des historiques immuables composés d'actions confidentielles et authentifiées. Cette solution repose uniquement sur l'utilisation de mécanismes de chiffrement et de hachage classiques, là où les technologies de type blockchain nécessitent également un consensus pour l'agrégation de blocs de transactions. Les résultats expérimentaux montrent que l'ajout d'actions dans une séquence de pair-actions est plus rapide qu'en utilisant une blockchain Ethereum, tout en provoquant un surplus en mémoire inférieur. De plus, les séquences de pair-actions n'impliquent aucune restriction sur les types de cycles de vie et d'artefacts manipulés ; elles permettent donc une abstraction complète de l'implémentation du contrôle de conformité. Ces éléments, combinés à d'autres,

font des séquences de pair-actions une alternative décentralisée, sécurisée, efficace et flexible à la blockchain et aux solutions existantes intervenant dans le BPM.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

Avant de plonger dans les considérations techniques de ce manuscrit, je souhaiterais manifester, dans les lignes qui suivent, ma gratitude envers les personnes qui ont participé à sa rédaction, parfois sans le savoir.

Tout d'abord, je tiens à remercier mon directeur Pr. Sylvain Hallé et mon codirecteur Pr. Raphaël Khoury, pour m'avoir donné l'opportunité d'effectuer cette thèse au sein du Laboratoire d'Informatique Formelle (LIF). Votre expertise, pédagogie, écoute et confiance m'ont apporté une grande sérénité dans la réalisation de ce travail. Avoir mené mes recherches sous votre supervision fut un véritable honneur, et je m'en estime particulièrement chanceux.

Je remercie également Pr. Benoit Montreuil du Georgia Institute of Technology, pour m'avoir permis d'effectuer une partie de mes travaux au Physical Internet Center (PI Lab), lors d'un stage riche en expériences et découvertes.

Je souhaite aussi remercier mes collègues de laboratoire, aussi bien au LIF qu'au PI Lab, et tout particulièrement Edmond La Chance, Massiva Roudjane, et Louis Faugère. Dans ce parcours, notre amitié et soutien mutuels ont constitué une grande force.

Merci également à mes amis au Canada et en France qui ont su, dans la limite du raisonnable, détourner mon esprit de cette tâche. En particulier, je remercie mes amis à Marseille, qui ont ensoleillé davantage encore les quelques séjours effectués dans ma ville natale.

Je voudrais aussi remercier ma petite amie, Manon, pour m'avoir encouragé tout au long de cette thèse, la plupart du temps à distance, et pour avoir fait preuve d'une patience quasi illimitée.

Enfin, merci aussi à mes frères et sœur, Clément, Maximilien, Laurent et Géraldine, pour leurs conseils avisés et leur partage d'expérience. Bien évidemment, je remercie sincèrement mes parents, Claude et Patrick, pour leur soutien indéfectible ; cette thèse n'aurait pu voir le jour sans leur accompagnement, manifesté à toutes les étapes de mes parcours personnel et professionnel.

INTRODUCTION

Par l'informatisation croissante de notre société, on assiste de nos jours à une abondance de données, créées et/ou manipulées par des opérateurs qui peuvent être nombreux et variés. Qu'il s'agisse d'employés de différents départements d'une entreprise coopérant pour délivrer un service, ou de centaines d'appareils connectés dans l'internet des objets (IoT) chargés de prélever des informations spécifiques, tous ont pour but de générer, modifier ou traiter des fragments de données.

Il existe un domaine qui a particulièrement profité de ce phénomène : celui des processus *d'affaires*, aussi appelés processus *opérationnels* (Dumas *et al.*, 2018). Ces derniers décrivent, entre autres, l'ensemble des tâches et activités nécessaires à la réalisation de services ou à la conception de produits. L'informatisation des documents et des processus a d'ailleurs provoqué la naissance d'un champ de recherche académique et d'un secteur industriel pleinement consacrés à la gestion des processus opérationnels (BPM, pour *Business Process Management*). Les travaux concernant le BPM ont amené plusieurs bénéfices au domaine des processus d'affaires, notamment par le biais de leur uniformisation, en standardisant leur séquencement par étapes, rendant ainsi possible l'automatisation de leur gestion. En conséquence, le BPM a grandement participé à l'amélioration de la performance, de la flexibilité et de la réactivité de l'implémentation des processus, ainsi que celles de leurs acteurs internes, particulièrement lorsqu'ils font face à des événements inattendus ou des changements de dernière minute.

Le BPM est un secteur en plein essor. Représentant un volume de 8,8 milliards USD en 2020, les estimations les plus récentes prévoient une croissance annuelle du marché de 10,5% dans les prochaines années, pour atteindre un volume de 14,4 milliards d'ici 2025 (MarketsandMarkets, 2020). D'après Grand View Research (2016), cette croissance est en partie due à la récente et progressive automatisation des processus dans le secteur médical

pour assurer une meilleure gestion des infrastructures hospitalières et monitorer plus précisément la prise en charge des patients. Cette évolution est particulièrement exacerbée par la transformation des systèmes de BPM en solutions SaaS (Software-as-a-Service) intégrées dans le *cloud*, qui offrent un accès facilité et une réduction globale des coûts opérationnels (Grand View Research, 2016; MarketsandMarkets, 2020). Parmi les solutions populaires, on peut mentionner l'écosystème NetWeaver [1] de l'entreprise SAP, qui à lui seul concentre plus de 15% des parts de marché des solutions de BPM (Datanyze, 2020).

Ainsi, le BPM devient un élément incontournable dans l'industrie. Un des avantages principaux du BPM pouvant expliquer cet intérêt croissant est qu'il permet la vérification de *conformité* des processus. Ceci signifie qu'il devient possible de s'assurer que, dans le cadre d'un processus complexe faisant intervenir de multiples entités ou *pairs*, les opérations diverses et variées sont effectuées dans le bon ordre et par les bonnes personnes. Cet aspect est effectivement crucial puisqu'il garantit l'exécution valide et légitime de processus d'affaires, tout en permettant d'identifier les *irrégularités* qui impacteraient négativement leurs performances ou provoqueraient des erreurs majeures dans leur exécution.

PROBLÉMATIQUE

De nos jours, l'usage de systèmes de BPM est largement répandu et leur légitimité bien établie. Néanmoins, dans la plupart des cas, de tels systèmes sont *centralisés*, ce qui signifie que la totalité des tâches composant un processus sont accomplies par l'intermédiaire d'un serveur ou organe central. Ce dernier est alors responsable de mettre en place les contrôles d'accès nécessaires, ainsi que de traiter et de stocker les manipulations effectuées. Bien que relativement « faciles » à mettre en place, ces systèmes centralisés présentent des inconvénients sur plusieurs niveaux.

1. https://www.sap.com/products/netweaver-platform.html

Premièrement, cette centralisation entraîne un manque majeur de flexibilité. En effet, dans le cadre de processus mettant en collaboration plusieurs entités (*p. ex.*, entreprises, institutions), il est nécessaire que ces dernières utilisent le *même* système et que chacun des acteurs qui les composent y aient accès. Deuxièmement, la centralisation de l'information peut s'avérer être une faille en elle-même, puisqu'il est plus aisé de *cibler* un système que l'on sait centralisé (qui peut stocker des données mais également être le garant de l'intégrité et de la confidentialité de celles-ci), plutôt que de compromettre des données qui sont potentiellement stockées sur plusieurs serveurs, à différents emplacements dans le monde, et dont l'intégrité et la confidentialité ne reposent pas sur les serveurs eux-mêmes mais sur d'autres mécanismes. Enfin, dans certains contextes, tels que celui de la *supply chain* (Christopher, 2016), centraliser le traitement et le stockage d'objets (virtuels ou physiques) circulant entre de multiples pairs, répartis aux quatre coins du globe, peut sembler contre-intuitif. Puisque ces objets passent de main en main, il pourrait sembler plus naturel que le pair manipulant l'objet à un instant donné réalise les diverses opérations directement sur celui-ci.

C'est notamment dans ce contexte qu'a eu lieu l'avènement de la modélisation de processus opérationnels centrés sur les artefacts (en anglais, *artifact-centric business process model*). Ce paradigme, introduit par Nigam & Caswell (2003), définit les processus opérationnels par des enchaînements d'opérations exprimées uniquement en termes de contraintes sur les *artefacts* (*p. ex.*, objets, documents) manipulés. Une séquence d'opérations est alors jugée valide tant que tout artefact intervenant dans le flux du processus respecte un ensemble de contraintes qui lui est propre : ces contraintes constituent le *cycle de vie* de l'artefact. Dans ce contexte, un artefact devient un objet *stateful*, c'est-à-dire un objet qui a conscience de son état. L'ensemble des modifications qui lui sont applicables peut être représenté par une multitude de formalismes dont, par exemple, les machines à états finis.

Transposé aux processus opérationnels centrés sur les artefacts, le contrôle de conformité signifie qu'une entité doit pouvoir garantir que le cycle de vie de chaque artefact impliqué est respecté en tout temps. Si ce domaine est relativement bien étudié dans une configuration où le contrôle est centralisé, une question reste néanmoins en suspens : comment s'assurer de la conformité d'un processus vis-à-vis du cycle de vie des artefacts qu'il manipule dans un contexte décentralisé ? C'est la question à laquelle cette thèse s'efforce de répondre.

CONTRIBUTIONS

Dans le milieu des méthodes de vérification formelle, assez peu de travaux sont consacrés aux solutions *totalement* décentralisées. En outre, ceux qui le sont, comme les propositions de Bauer & Falcone (2016) et Hallé (2013), montrent des limites de flexibilité dans la spécification du cycle de vie ou dans les méthodes de communication. Plus problématique encore, elles supposent l'existence d'une certaine forme de confiance ; celle-ci est présente entre les composants d'un même système pour Bauer & Falcone (2016), ou seulement d'un client envers un serveur pour Hallé (2013). L'objet de cette thèse est donc de proposer une décentralisation totale de la vérification de conformité de processus, qui soit à la fois flexible et sécurisée, en se basant sur deux approches.

VÉRIFICATION DÉCENTRALISÉE BASÉE SUR UNE BLOCKCHAIN

La première méthode repose sur la technologie des *blockchains*, qui a été introduite par Nakamoto (2008) avec une preuve d'application devenue incontournable depuis : le Bitcoin. Le principe de la blockchain est de construire un registre de transactions financières authentifié, transparent, immuable, et partagé par tous ses utilisateurs. Elle utilise des notions de cryptographie et de réseaux pair-à-pair (P2P, pour *Peer-to-Peer*) pour échanger et stocker

des *blocs de transactions signées* à travers un réseau de *nœuds*. Les blocs sont reliés entre eux par des *valeurs de hachage*, ou *condensats* ; ainsi, chaque nœud référence le suivant en incorporant le condensat de ce dernier dans son en-tête. Le choix du bloc à joindre à la blockchain actuelle se fait à l'aide d'un consensus sur lequel les nœuds se mettent d'accord au préalable. Depuis son invention, la blockchain a beaucoup évolué et a été appliquée dans bon nombre de domaines, tels que le secteur de l'e-santé (Agbo *et al.*, 2019), des maisons intelligentes (Dorri *et al.*, 2017a), des véhicules connectés (Dorri *et al.*, 2017b), ou encore des processus opérationnels (Viriyasitavat *et al.*, 2018).

Notre première contribution a donc été d'utiliser la blockchain comme réceptacle des manipulations sur les artefacts et de considérer les transactions qu'elle contient comme une séquence d'événements alimentant les moniteurs dédiés à la vérification de propriétés. Les manipulations sont réalisées par l'intermédiaire de *smart contracts*, des programmes partagés et exécutés par tous les pairs du réseau via une blockchain Ethereum (Wood, 2019), la première plateforme ayant permis l'implémentation de *smart contracts*. En outre, nous implémentons cette architecture à des domaines d'application concrets, à savoir l'Internet Physique et la logistique hyperconnectée. Ces paradigmes s'inspirent des notions et composants fondamentaux d'internet pour améliorer globalement la *supply chain* et la logistique ; ces améliorations concernent, par exemple, l'optimisation de la livraison des biens ainsi que la réduction des impacts environnementaux et sociaux. Ainsi, nous définissons et implémentons une série de propriétés et contraintes de cycle de vie de colis à livrer, qui nous semblent pertinentes dans un contexte de logistique hyperconnectée. L'ensemble de ces implémentations est évalué expérimentalement à travers une simulation AnyLogic [2] de logistique hyperconnectée.

Ces travaux ont donné lieu à deux publications : l'article de journal « Improving Hyperconnected Logistics With Blockchains and Smart Contracts » (Betti *et al.*, 2019), et le chapitre

2. https://www.anylogic.com/

de livre « Smart Contracts-Enabled Simulation for Hyperconnected Logistics » (Betti *et al.*, 2020), tous deux coécrits avec mes codirecteurs, les professeurs Sylvain Hallé et Raphaël Khoury, ainsi que Benoit Montreuil, professeur au Georgia Institute of Technology.

LES SÉQUENCES DE PAIR-ACTIONS

Comme nos résultats expérimentaux le montrent, l'approche par blockchain implique de nombreuses « lourdeurs », la principale étant le taux de transactions par seconde (TPS) du réseau insuffisant ; cette insuffisance est exacerbée dans un scénario aussi ambitieux que celui de la logistique hyperconnectée. De plus, certaines notions fondamentales, telles que les frais de transactions ou les limitations de ressources calculatoires allouées aux *smart contracts* ont peu de sens dans la mise en place de processus collaboratifs entre, par exemple, des entreprises et/ou institutions.

Pour ces raisons, nous introduisons une approche originale où l'historique d'actions n'est plus stocké dans une blockchain mais dans l'artefact lui-même. Ces historiques sont confidentiels, authentifiés, immuables et sont appelés *séquences de pair-actions*. Le fonctionnement de ces dernières s'inspirent des blockchains en ceci que les *pair-actions*, c'est-à-dire les événements atomiques de la séquence, sont chaînées par leur condensat. Cependant, ce dernier s'éloigne de la définition classique des condensats puisqu'il est *signé* par le pair réalisant l'action, permettant de l'authentifier. Une autre différence importante est que chaque pair effectue une action au nom d'un *groupe*. La notion de groupe est similaire à celle des *rôles* dans le Role-Based Access Control (Ferraiolo & Kuhn, 1992), auxquels on associe des utilisateurs et des tâches autorisées. En assignant une clé symétrique à chaque groupe, il devient alors possible pour le pair de chiffrer l'action réalisée pour que le cryptogramme résultant soit déchiffrable uniquement par les pairs du groupe correspondant, ce qui rend les

actions complètement confidentielles. De plus, aucun consensus n'y est nécessaire puisque les pair-actions sont ajoutées successivement à la séquence par les pairs manipulant l'artefact ; ceci est rendu possible par le fait que la séquence accompagne l'artefact en tout temps sous la forme de méta-données. Bien que l'application des séquences de pair-actions ne s'y limite pas, nous définissons formellement les artefacts considérés dans cette thèse et les contraintes de cycles de vie exprimables sur ces derniers.

À titre d'illustration, nous présentons un cas d'application des séquences de pair-actions et de la vérification de contraintes de cycle de vie dans le cadre de manipulations de données médicales, effectuées par un ensemble de pairs, à savoir les patients, le personnel soignant, et les assureurs. De plus, nous avons implémenté ces séquences de pair-actions dans une bibliothèque open source et nous les intégrons dans un logiciel de manipulation de fichiers PDF échangés entre pairs. Ainsi, il devient possible d'extraire une séquence stockée dans un fichier PDF, d'en déchiffrer les pair-actions appropriées, et d'y ajouter des pair-actions.

Ces travaux ont donné lieu a un article de journal, titré « Decentralised enforcement of document lifecycle constraints » (Hallé *et al.*, 2018b), coécrit avec les professeurs Sylvain Hallé et Raphaël Khoury, ainsi qu'Yliès Falcone et Antoine El-Hokayem, respectivement professeur et post-doctorant à l'Université Grenoble-Alpes.

Par l'intermédiaire d'une amélioration de l'implémentation présentée dans cette dernière publication, nos résultats expérimentaux montrent que cette approche est plus *légère* que celle de la blockchain, et qu'elle semble donc plus adaptée à certains scénarios. Concrètement, cela signifie qu'elle permet d'ajouter plus d'actions que la blockchain dans le même temps, tout en diminuant le surplus de données en mémoire, intrinsèque à la construction de l'historique et nécessaire à sa sécurité.

LIEU DU TRAVAIL

La majorité des travaux décrits dans ce document ont été effectués au sein du Laboratoire d'Informatique Formelle [3] (LIF), un laboratoire rattaché au département d'informatique et de mathématique de l'Université du Québec à Chicoutimi (UQAC), au Canada. Il est dirigé par le professeur Sylvain Hallé, cofondateur du LIF et détenteur d'une chaire de recherche du Canada sur la spécification, la mise à l'essai, et la vérification de logiciels. Le LIF est spécialisé dans le développement et l'application de méthodes formelles pour les systèmes informatiques, notamment dans les domaines de la sécurité, la vérification et le test logiciels.

Néanmoins, la victoire au concours de bourse « Stages internationaux – Énergie-Numérique-Aérospatiale » 2017-2018 du Fonds de recherche du Québec – Nature et technologies (FRQNT) m'a permis de réaliser une partie de mon doctorat au sein du Physical Internet Center [4] (aussi appelé PI Lab), un laboratoire rattaché au Supply Chain & Logistics Institute du Georgia Institute of Technology (Georgia Tech), aux États-Unis. Ce séjour a été effectué sous la forme d'un stage de six mois de mars à octobre 2018, sous la supervision du professeur Benoit Montreuil. Pr. Montreuil est le pionnier des principes du Physical Internet, le fondateur ainsi que le directeur du Physical Internet Center, et le détenteur d'une chaire Coca-Cola « Material Handling and Distribution ». Le PI Lab est principalement axé sur le développement et la mise en application des principes du Physical Internet et de la logistique hyperconnectée à des situations réelles, le tout en partenariat avec les entreprises du secteur de la *supply chain* et les institutions concernées.

---

3. https://liflab.ca/
4. https://www.picenter.gatech.edu/

ORGANISATION DU DOCUMENT

Le contenu de la présente thèse est réparti en six chapitres, organisés comme suit. Le premier chapitre détaille les notions d'artefact et de cycle de vie ; celles-ci sont par ailleurs illustrées par des scénarios concrets dont, entre autres, un premier exemple concernant la manipulation de données médicales et un second appliqué à la livraison de colis dans un contexte de logistique hyperconnectée. Ces deux principaux exemples seront ensuite réutilisés par la suite pour illustrer les approches considérées dans ce document.

Ensuite, le second chapitre présente les travaux existants traitant de la spécification et la vérification de cycle de vie. Nous étudions ici les différentes formes de représentation des propriétés de cycle de vie, telles que les automates finis ou les logiques temporelles, et nous listons l'ensemble des travaux proposant des solutions pour vérifier la conformité de processus vis-à-vis de ces spécifications. Nous verrons que ces solutions, principalement basées sur les notions de *runtime verification* et de *model checking* peuvent être réparties en trois catégories : les approches centralisées, celles par vérification statique, et enfin, les décentralisées. Nous en concluons notamment le manque de solutions permettant la décentralisation complète du processus de vérification.

Parmi les solutions permettant la mise en place de processus collaboratifs décentralisés et sécurisés, la technologie des blockchains paraît particulièrement prometteuse. Dans le Chapitre 3, nous détaillons donc le fonctionnement de cette dernière et ses principes fondamentaux. Étant donnés la variété et le nombre important de blockchains existantes, nous réalisons cette description par l'intermédiaire de la plateforme Ethereum, la première blockchain ayant permis le déploiement de *smart contracts*.

Une fois ces notions établies, nous utilisons dans le Chapitre 4 une blockchain Ethereum pour stocker les différentes actions émises par les agents d'une simulation AnyLogic de logis-

tique hyperconnectée. Celles-ci sont intégrées dans la blockchain par l'intermédiaire de *smart contracts*, que nous détaillons. Puis, nous listons un ensemble de contraintes et de propriétés de cycle de vie de colis à livrer. En outre, nous implémentons ces composantes de cycle de vie à l'aide de BeepBeep (Hallé, 2018), une bibliothèque Java open source de *Complex Event Processing*. Notre blockchain Ethereum permet ainsi de générer des événements alimentant les moniteurs propres aux propriétés présentées, et d'effectuer ainsi la vérification par runtime monitoring de façon décentralisée.

Les résultats expérimentaux mettant en défaut l'approche par blockchain, nous introduisons, au Chapitre 5, l'approche originale des séquences de pair-actions, permettant de construire des historiques partagés, confidentiels, authentifiés et immuables. Ceci est possible par l'unique utilisation de mécanismes de chiffrement et de fonctions de hachage. En outre, nous formalisons la définition et la structure des artefacts considérés ici (que nous appelons alors « documents »), et présentons les contraintes de cycle de vie associées à ces artefacts. L'exemple concret de manipulations de données médicales vu au Chapitre 1 est alors revisité pour illustrer l'approche par séquence de pair-actions.

Enfin, dans le Chapitre 6, nous compilons l'ensemble des résultats expérimentaux. Dans un premier temps, ceux-ci concernent les performances générales de l'approche par blockchain et de la vérification des propriétés énumérées au Chapitre 4. Dans un second temps, nous présentons les résultats liés aux implémentations des séquences de pair-actions développées et les divers outils associés. Puis, nous concluons par une comparaison exhaustive des deux approches sur un ensemble de critères allant des performances globales à la sécurité de chaque approche, en passant par leur applicabilité.

Bien que les implémentations présentées ne soient pas adaptées aux scénarios où des actions sont réalisées *en simultané* sur un même artefact, nous montrons que les séquences

de pair-actions représentent une alternative crédible aux blockchains actuelles, dans certains contextes d'utilisation. En outre, elles fournissent un traitement plus rapide des opérations, avec une réduction du surplus en mémoire lié à la construction de l'historique, tout en permettant une isolation des artefacts les uns par rapport aux autres. Ces éléments, entre autres, montrent que les séquences de pair-actions constituent une solution performante et flexible pour la réalisation de contrôles de conformité décentralisés.

CHAPITRE I

ARTEFACTS ET CYCLES DE VIE

Le but de ce chapitre est de donner un contexte et une motivation au travail présenté dans cette thèse. Nous y décrivons ainsi en détail deux scénarios provenant de domaines variés, tirés de la gestion de documents médicaux et de l'Internet Physique. En outre, ceux-ci seront l'objet d'applications des approches de conformité de cycle de vie par blockchain et par séquence de pair-actions plus loin dans cette thèse, respectivement aux chapitres 4 et 5. Nous présentons ensuite, plus rapidement, quelques exemples tirés de la littérature dont les domaines d'applications sont, là aussi, très diversifiés. Enfin, nous concluons ce chapitre en identifiant les points communs à tous ces exemples ; ces derniers gravitent autour des notions d'artefact et de respect de cycle de vie, qui constituent le point central sur lequel est appuyée la contribution de cette thèse.

1.1 PREMIER SCÉNARIO : GESTION DE DOCUMENTS MÉDICAUX

Nous présentons tout d'abord un scénario, adapté de la littérature (Bielova *et al.*, 2009; Bielova & Massacci, 2011), concernant la gestion de dossiers médicaux traitant de la prise en charge des patients et de l'évolution de leurs soins au sein d'un hôpital. Comme nous l'avons vu en introduction, l'informatisation des données médicales fait l'objet d'une application massive de BPM. En effet, celui-ci permet un suivi plus efficace et précis des patients, qui sont parfois amenés à consulter plusieurs médecins exerçant leur activité dans différents hôpitaux, cliniques ou centres de soins. Dans ce type de contexte, un dossier peut être vu comme un document passant de main en main entre le patient, le personnel médical, et, potentiellement, des intervenants extérieurs tels que la compagnie d'assurance du patient et le régime de santé publique de la région ou du pays concerné.

Ici, le but est de montrer que les manipulations d'un tel document doivent être intrinsèquement soumises à des contraintes, et ce afin de préserver la confidentialité des informations qu'il contient ainsi que d'effectuer un suivi des soins efficace et respectueux de la déontologie médicale.

1.1.1 STRUCTURE DU DOCUMENT

Il est tout d'abord nécessaire de définir la nature et la structure d'un tel document. On peut ainsi considérer que celui-ci doit comporter plusieurs types d'informations, notamment :

— les identifiants du patient,

— un numéro d'assurance,

— une série de tests, demandée par le médecin et réalisée par du personnel médical,

— des ordonnances de médicaments, remplies par le pharmacien avec l'approbation de la compagnie d'assurance du patient

Chacune de ces informations peut être saisie dans le document soit en tant que valeur atomique ou comme chaîne de caractères (le numéro d'assurance et les identifiants du patient), soit comme une liste (les tests), soit comme un ensemble de paires clé-valeur (une *mappe*). Ce dernier cas s'applique aux tests pris individuellement, puisqu'ils associent une requête de test à ses résultats.

1.1.2 TYPES DE CONTRAINTES

Le patient, les médecins, les pharmaciens, les autres membres du personnel médical ainsi que la compagnie d'assurance du patient peuvent accéder à ce document ; cependant chacun possède des droits d'accès distincts. Étant donné la nature confidentielle d'un tel document, il va sans dire que les entités sont sujettes à différents privilèges pour lire, écrire, ou

modifier les parties du document, et ce afin d'en assurer le bon usage. Ces contraintes peuvent être partagées en trois catégories, à savoir les contraintes de contrôle d'accès, d'intégrité, et d'ordonnancement.

Parmi les contraintes d'accès, nous mentionnons les suivantes :

— certaines parties du document ne doivent pas être vues par certains pairs. Par exemple, le médecin ne peut pas lire le numéro d'assurance, tandis que la compagnie d'assurance pourrait ne pas avoir accès à certaines informations médicales sensibles ;

— certaines parties du document ne peuvent être modifiées par certains pairs. Par exemple, le pharmacien peut lire les prescriptions, mais n'est pas autorisé à les altérer ;

— certaines actions ne sont pas autorisées à certains pairs. Par exemple, une infirmière ne peut en aucun cas écrire une prescription ;

— d'autres intervenants peuvent également avoir un accès limité au document. Par exemple, lors de tests de médicaments, des chercheurs pourraient récupérer les données concernant les potentiels effets secondaires. Aussi, des instances gouvernementales pourraient accéder à ce fichier et à d'autres du même type afin de récolter des données anonymes sur la propagation de certaines maladies ;

De leur côté, les *contraintes d'intégrité* imposent des restrictions sur les valeurs qui peuvent être saisies dans chaque champ du document. Ceci inclut, par exemple, la condition que le nom d'un médicament prescrit soit sélectionné dans une liste de médicaments approuvée par le gouvernement, ou que le dosage mensuel total soit égal à la somme des dosages quotidiens individuels.

Enfin, les *contraintes d'ordonnancement* expriment des restrictions sur l'ordre dans lequel des actions peuvent être réalisées afin d'être valides. Si une action ne respecte pas cet ordre (on dit qu'elle est « *out-of-order* »), le document sera toujours lisible, mais une

vérification de son historique révélera son état non conforme. Ainsi, une contrainte possible pourrait être qu'une prescription soit nécessairement validée par la compagnie d'assurance avant qu'elle puisse être remplie par le pharmacien, ou que les requêtes de tests médicaux de la part du médecin soient exécutées avant que toute autre action puisse être réalisée sur le document.

1.1.3 ACTIONS ATOMIQUES

Dans ce scénario, comme dans beaucoup d'autres situations, nous pouvons supposer que le document peut être modifié uniquement au moyen d'un nombre restreint d'actions atomiques incluant les actions `write`, `read` et `update`, qui permettent respectivement d'ajouter une valeur dans un champ spécifique du document, de lire celui-ci, et de modifier sa valeur. Le document supporte également un certain nombre d'actions spécifiques à son usage, telles que `perform`, `approve` et `fill`. L'action `perform` indique qu'un test médical demandé par le médecin a été effectué, et permet donc à l'infirmière qui s'en est chargée d'agréger les résultats dans la partie du test correspondant. L'action `approve` est accomplie par l'employé de la compagnie d'assurance pour indiquer que le coût de la prescription est remboursé. Enfin, `fill` est une action effectuée par le pharmacien lorsqu'il exécute l'ordonnance. En pratique, ces actions additionnelles ne sont pas strictement nécessaires. En effet, certains de leurs comportements pourraient être exprimés à l'aide des actions basiques `read`, `write` et `update`, et en y associant des contraintes sur les parties du document manipulées. Cependant, ces actions supplémentaires permettront plus tard d'exprimer le cycle de vie de façon plus concise.

1.1.4 EXEMPLE DE SCÉNARIO COMPLET

Le cycle de vie impose également qu'aucune action ne peut être exécutée tant que le document n'a pas été initialisé, en remplissant tout d'abord une section d'informations personnelles sur le patient, puis en inscrivant son numéro d'assurance. Ce n'est que lorsque ces deux étapes sont réalisées que le document peut commencer à être utilisé. Nous supposons que le document est créé par l'hôpital, initialisé avec les informations personnelles du patient par lui-même et que son numéro d'assurance est renseigné par l'infirmière.

Une fois cela fait, le médecin peut effectuer deux actions : prescrire un médicament, ou demander qu'un test médical soit effectué. Demander un test se fait à l'aide de l'action `test`, qui indique qu'une certaine valeur *v* caractérisant le test a été saisie dans la section *test*. À la suite de cela, aucune action ne peut être faite sur le document tant que le test demandé n'a pas été réalisé, ce qui sera enregistré dans le document à l'aide de l'action spécifique `perform` expliquée plus tôt. Autrement, si le médecin prescrit directement un médicament, celui-ci doit d'abord être approuvé par l'assurance avant de pouvoir être remplie par le pharmacien. À toute étape du processus, le patient peut rapporter des effets secondaires, qui seront écrits dans le document. Si un effet secondaire est signalé, aucune action ne peut être réalisée tant que le médecin n'a pris connaissance de celui-ci en utilisant l'action `read`.

L'ensemble de ces politiques constituent un exemple crédible de cycle de vie dans un contexte de gestions de données médicales, et une partie de cette thèse a comme objectif de montrer comment notre approche par séquence de pair-actions, introduite au Chapitre 5, permet de faire respecter des contraintes de cette nature de façon décentralisée.

1.2 DEUXIÈME SCÉNARIO : INTERNET PHYSIQUE ET LOGISTIQUE HYPERCONNECTÉE

Notre deuxième scénario est axé autour de la gestion de la chaîne logistique. Celle-ci implique, entre autres, le déploiement de marchandises à travers différents marchés géographiques et leur livraison sur de courtes et longues distances. Depuis 2011, l'Internet Physique théorisé par Montreuil (2011) propose notamment une transformation du traitement, stockage, et transport des marchandises vers une gestion de la chaîne d'approvisionnement (ou *supply chain*) plus efficace en termes de temps de livraison, de coût, ainsi que d'impacts sociaux et environnementaux. Ce domaine a vu émerger celui de la logistique hyperconnectée, une refonte radicale des réseaux classiques de la *supply chain*, où chaque colis à livrer est traité comme un paquet IP *physique* qui peut transiter à travers différents *hubs* (*p. ex.*, points de livraison, plateformes de stockage, centres de tri).

Récemment, le concept de logistique hyperconnectée a été couplé avec les logistiques urbaines pour transformer radicalement la façon dont les marchandises physiques circulent dans un environnement urbain, faisant place à la logistique urbaine hyperconnectée (Crainic & Montreuil, 2016). Ces nouveaux paradigmes ont donné naissance à un système multipartite complètement distribué où chaque colis porte un fragment de l'état de la *supply chain* lors de ses déplacements autour du globe. Dans un tel contexte, s'assurer que l'entièreté de la chaîne demeure dans un état cohérent et détecter divers types d'erreurs devient une tâche complexe.

Dans cette section, il s'agit donc de présenter les notions d'Internet Physique et de logistique hyperconnectée, puis d'exposer de façon succincte des contraintes à appliquer à l'acheminement de colis dans un cadre de logistique hyperconnectée. Le Chapitre 4 approfondit ce scénario et donne un formalisme complet de plusieurs propriétés à respecter dans ce contexte.

1.2.1 LES PRINCIPES DE L'INTERNET PHYSIQUE

Le but de l'Internet Physique, introduit par Montreuil (2011), est d'augmenter de manière significative l'efficacité, la capacité et la durabilité des services répondant aux demandes de la société en termes de marchandises physiques. Ceci inclut les manières de déplacer, déployer, réaliser, approvisionner, concevoir et utiliser ces dernières. L'Internet Physique a été formellement défini comme un système de logistique hyperconnectée établissant un partage des ressources ouvert et transparent ainsi qu'une consolidation des flux via une encapsulation, une modularisation, des protocoles et des interfaces standardisés. Ce système est dit « hyperconnecté » car ses composants et acteurs sont intensément interconnectés sur plusieurs couches, partout et en tout temps. Les couches d'interconnexions de l'Internet Physique incluent les domaines numérique, physique, opérationnel, d'affaires, légaux et personnels.

Selon Montreuil (2011), les composants clés pour une implémentation intégrale, une adoption à grande échelle et une exploitation internationale de l'Internet Physique sont : 1) un ensemble homogène de conteneurs logistiques modulaires et standardisés ; 2) des équipements et technologies adaptés à ces conteneurs ; 3) des protocoles de logistiques standardisés ; 4) des installations et parcours ouverts et certifiés ; 5) des systèmes de supervision de la logistique dans sa globalité ; 6) des plateformes transactionnelles et décisionnelles ouvertes ; 7) des analyses, optimisations et simulations intelligentes, orientées données ; 8) des fournisseurs de services logistiques transparents et certifiés.

À terme, Montreuil (2011) envisage que l'Internet numérique, l'Internet des Objets (IoT), l'Internet Physique et l'Internet de l'Énergie se développeront dans le contexte d'infrastructures hyperconnectées qui seront des piliers économiques et sociaux qui, par leur

synergie, amélioreront leur capacité, leur efficacité et leur durabilité, tout en réduisant de façon dramatique leurs coûts marginaux d'exploitation, tel que soutenu par Rifkin (2014).

1.2.2 LE PARADIGME DES LOGISTIQUES HYPERCONNECTÉES

Dans le modèle hyperconnecté défini par Montreuil *et al.* (2018), le monde entier peut être divisé, à la plus petite échelle, en *unit zone* (zone unitaire), dont la taille dépend de la densité en demandes (nombre de colis à livrer/poster). Les *unit zones* adjacentes sont groupées en *local cells* (cellules locales), qui sont à leur tour groupées en *areas* (aires), qui forment elles-mêmes des *regions* (régions). En parallèle, plusieurs couches de réseaux de *hubs* (c.-à-d., centres de tri par lesquels transitent les paquets, ou servant de points de livraison) sont définies : les *access hubs* lient les *unit zones* ensemble, les *local hubs* lient les *local cells* entre elles, et les *gateway hubs* lient les *areas*. Différents niveaux de *hubs* peuvent exister au sein d'une même entité physique (*p. ex.*, un *local hub* peut également être un *access hub*), permettant ainsi de lier les différentes couches entre elles.

Le concept de logistique hyperconnectée introduit plusieurs ruptures par rapport aux logistiques de *supply chain* traditionnelles. Par exemple, usuellement un conteneur qui doit voyager sur de longues distances *dans une ville* entre un point A et un point B sera chargé dans un camion qui réalisera le voyage du début à la fin. Dans la logistique hyperconnectée, au contraire, ce même conteneur « sautera » à maintes reprises entre des *access hubs* et sera transféré d'un camion à un autre jusqu'à atteindre sa destination finale. Lors du passage d'une couche du réseau hyperconnecté à une autre, le contenu du conteneur pourra même être divisé en plus petits paquets, chacun ayant sa propre séquence de *hubs* à visiter avant d'atteindre sa destination finale.

1.2.3 CONTRAINTES SUR LES MANIPULATIONS DE COLIS

De façon similaire au premier scénario vu dans la section précédente, l'acheminement de colis doit être sujet à différentes contraintes ; ceci est fait dans le but d'obtenir un meilleur contrôle sur le processus de livraison et, par conséquent, d'en fournir une exécution plus efficace, avec le moins d'erreurs ou d'irrégularités possible. Comme en Section 1.1, ces contraintes peuvent être réparties en contraintes de contrôle d'accès, d'intégrité et d'ordonnancement.

Par exemple, concernant les droits d'accès, nous pouvons estimer que seul le client puisse modifier la destination finale d'un colis ; ainsi un livreur ou un employé d'un centre de stockage ne pourrait pas dérouter un colis, intentionnellement ou par erreur. On peut aussi énoncer que les actions de livraison ou de ramassage ne peuvent être effectuées que par des livreurs, qu'ils soient conducteurs de camion ou de véhicules plus légers (*p. ex.*, camionnette, vélo, ou petits véhicules électriques circulant au sein d'une ville). Cette propriété permet d'exclure la possibilité qu'un colis soit ramassé par une personne illégitime qui tenterait de le dérober. Aussi, seul le client devrait être capable de confirmer la réception d'un colis lorsque celui-ci doit être délivré en main propre, pour que personne ne puisse le faire à sa place ; cette contrainte permet ainsi d'éviter qu'un colis soit prétendument livré alors que le destinataire ne l'a jamais reçu.

D'autre part, une contrainte d'intégrité qui nous paraît particulièrement essentielle est qu'un colis devrait toujours se rapprocher de sa destination finale, à un seuil de distance près. Cette dernière nuance permet de prendre en compte le cas où le colis est envoyé à un *hub* légèrement plus éloigné (relativement à la distance totale à parcourir) au début de son acheminement. Un autre exemple pourrait être que la localisation d'un ramassage devrait toujours correspondre au lieu de la livraison précédente, évitant ainsi des « téléportations

de colis », qui pourraient être dues à des erreurs de mise à jour des informations ou à des tentatives de vol de colis.

Enfin, une première contrainte d'ordonnancement légitime serait que deux livraisons intermédiaires d'un colis soit nécessairement séparées par un ramassage, et inversement. En effet, une telle situation signifierait qu'une ou plusieurs manipulations n'ont pas été correctement renseignées dans le système ; ceci pourrait exposer, par exemple, plusieurs défaillances de matériel de suivi (*p. ex.*, scanners, serveurs). Il est également intéressant de s'assurer qu'une livraison finale par un livreur soit toujours suivie d'une confirmation de réception de la part du client. On pourrait d'ailleurs estimer que cette confirmation doit être faite dans un intervalle de temps spécifique : s'il n'y a toujours pas de confirmation, par exemple, deux heures après la livraison finale, on pourrait en conclure que le client n'a jamais reçu son colis.

Là encore, l'ensemble de ces contraintes représente un exemple crédible de cycle de vie d'un scénario réel et concret, cette fois-ci dans un contexte de livraison de colis. Dans le cadre de la logistique hyperconnectée, la vérification de la conformité de ce type de cycle de vie est un objectif particulièrement ambitieux compte tenu de la multiplication des manipulations réalisées et des acteurs impliqués. Un des points centraux de cette thèse est, en outre, de proposer une solution qui permette de monitorer ces manipulations et d'en contrôler la conformité, tout en étant adaptée à la décentralisation induite par la logistique hyperconnectée ; celle-ci est introduite dans le Chapitre 4 et repose sur les technologies de type *blockchain*.

1.3 AUTRES SCÉNARIOS

Dans cette section, nous listons d'autres scénarios pour lesquels les notions de BPM, de respect de contraintes ou de politiques peuvent être appliquées. Ceux-ci s'étalent sur des

domaines très diverses, allant de la gestion de demandes de prêts bancaires à l'évaluation de performances sportives.

1.3.1 PROCESSUS BANCAIRES

Un contexte dans lequel l'ordonnancement des manipulations sur un document est particulièrement sensible est le domaine bancaire. Dans ce cas, contraindre le flux des opérations possibles permet d'imposer que les lois et règles du milieu concerné soient forcément respectées, avec les précautions assurant une gestion financière avisée. Ce scénario a récemment été étudié par Mukkamala *et al.* (2008) et un nouveau formalisme pour exprimer les restrictions régissant les flux de document y est proposé. Ce formalisme, la *Process Matrix* (matrice de processus), est strictement plus expressif que le BPMN (*Business Process Model and Notation*, une méthode de représentation graphique des processus d'affaires que nous détaillons au Chapitre 2) puisque qu'elle permet aux utilisateurs de décrire des restrictions conditionnelles sur l'obligation de réaliser certaines étapes.

Le Tableau 1.1 décrit l'exemple utilisé par les auteurs, à savoir un processus de demande de prêt. Chaque ligne du tableau indique une activité du processus, listée dans la première colonne. Les trois colonnes suivantes définissent les droits d'accès pour chacun des trois rôles (*App* pour le demandeur, *CW* pour le chargé du dossier, *Mgr* pour le gestionnaire) qu'un sujet peut avoir dans ce processus. Par exemple, le demandeur peut écrire une demande, qui peut être ensuite lue par le chargé du dossier et le gestionnaire, mais seul ce dernier peut donner la deuxième approbation pour la demande. La colonne suivante liste les contraintes sur l'ordonnancement des activités. Elle fait la distinction entre les prédécesseurs *normaux* et les prédécesseurs *logiques*, ces derniers étant indiqués par un astérisque (*). Si une activité A est un prédécesseur logique de l'activité B, alors à chaque ré-exécution de l'activité A, l'activité B doit être ré-exécutée elle aussi ; si A est un prédécesseur normal de B, alors A peut

	Activities	Roles			Prede-cessors	Activity Condition
		App	CW	Mgr		
1	Application	W	R	R		
2	Register Customer Info	W	W	W		
3	Approval 1	W	W	R	$\overset{*}{1}$,2	
4	Approval 2	D	R	W	$\overset{*}{1}$,2	¬Rich
5	Payment	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	¬Hurry∧ Accept
6	Express Payment	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	Hurry∧ Accept
7	Rejection	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	¬Accept
8	Archive	D	W	R	$\overset{*}{5}$,$\overset{*}{6}$,$\overset{*}{7}$	

TABLEAU 1.1 : Process Matrix pour une demande de prêt, par Mukkamala *et al.* (2008)
© 2011 IEEE.

être ré-exécutée sans qu'il soit nécessaire de ré-exécuter B. La dernière colonne décrit une condition booléenne optionnelle qui pourrait rendre l'activité superflue. Dans cet exemple, la seconde approbation peut être omise si le prédicat *Rich* est vrai.

1.3.2 POLITIQUES D'INTÉGRITÉ DES DONNÉES

Dans d'autres cas, il peut être intéressant de restreindre les manipulations de données à des actions bien spécifiques. Par exemple, les *assured pipelines* de Boebert & Kain (1985) facilitent le transfert sécurisé d'informations confidentielles entre des entités fiables en spécifiant quelle transformation des données doit être réalisée avant tout autre traitement de celles-ci. Par exemple, des *assured pipelines* peuvent être utilisés pour assurer que des données

confidentielles soient anonymisées avant d'être diffusées, ou que la saisie effectuée par un utilisateur soit formatée avant d'entrer dans un système.

Les restrictions de manipulations peuvent s'appliquer aux types d'actions mais également aux objets eux-mêmes, selon certaines de leur caractéristique. Dans ce sens, on peut mentionner le modèle de Brewer & Nash (1989), la *Chinese Wall policy*, qui peut être mis en place pour empêcher des conflits d'intérêts d'avoir lieu. Par exemple, appliquer une telle politique pourrait empêcher un consultant de conseiller deux entreprises concurrentes, ou un investisseur de suggérer des placements dans une compagnie dans laquelle il a des parts. Dans ce modèle, un utilisateur qui accède à un objet de données o est interdit d'accès aux autres objets de données qui sont en conflit avec o.

Dans cette lignée, Sobel & Alves-Foss (1999) proposent une application du modèle de *Chinese Wall* basé sur les traces d'événements, enrichie par des notions pertinentes de *data-relinquishing* (abandon des données) et d'intervalle de temps. Dans ce cadre, chaque objet o est associé à une liste de paires action-opérateur, et les actions (`create` ou `read`) que les opérateur ont exécutées sur l'objet sont séquentiellement listées. La politique est exprimée comme un ensemble de conflits d'intérêts $\mathscr{C}$; chaque objet O_i est associé à ses conflits d'intérêts $\mathscr{C}_i$, qui liste les autres objets qui sont en conflit. Le respect de la *Chinese Wall policy* est assuré en empêchant tout utilisateur ayant accédé à un objet dans l'ensemble $\mathscr{C}_i$ d'accéder à l'objet O_i.

Enfin, la *low-water-mark policy*, conçue par Biba (1977), est un modèle dans lequel chaque objet de données est associé à un niveau d'intégrité indiquant sa fiabilité. On peut contraindre les actions réalisables par un sujet, en imposant qu'il ne puisse écrire que dans les objets dont le niveau d'intégrité est inférieur ou égal au sien, et qu'il ne puisse lire que les

objets dont le niveau d'intégrité est supérieur ou égal au sien. Cela évite que des sujets et des objets soient infectés avec des données (de bas niveau d'intégrité) peu ou pas fiables.

1.3.3 CARTES À PUCE

Certaines cartes à puces, telles que les MIFARE Classic [5], sont utilisées pour accorder l'accès aux transports en commun, notamment à Londres avec la carte Oyster (The London Pass, 2020). Ces cartes enregistrent le nombre de voyages restants ainsi qu'une trace des précédents voyages dans le système. La carte est éditée par un *lecteur de cartes*. Des cartes similaires sont utilisées dans d'autres contextes telles que les cartes de bibliothèque, les cartes d'hôtel, les cartes de membre, de sécurité sociale, de location de voiture et d'accès à des parcs d'attractions ou musées.

Dans un tel cas, le « document » est physique, et est porté d'un lecteur à un autre lorsque le passager voyage à travers le réseau de transport. Cependant, dans ce contexte, la source de méfiance n'est pas les lecteurs, mais le transporteur de la carte. Par exemple, il n'est pas souhaitable que la même carte entre deux fois de suite dans la même station, ce qui indiquerait probablement une tentative d'utilisation de la même carte pour deux voyageurs. Les informations contenues dans la carte MIFARE pourraient aussi être utilisées afin d'autoriser ou interdire le transfert d'un réseau de transport à un autre, avec toute autre restriction décrite par la politique de cycle de vie.

1.3.4 DONNÉES SPORTIVES

Johansen *et al.* (2015) ont développé un cas d'utilisation spécifique dans une équipe de sport de haut niveau. En effet, l'impact de l'analyse des données sportives sur les performances

5. http://www.mifare.net

est bien reconnue et implique un besoin croissant d'enregistrements de données techniques, médicales et personnelles sur les athlètes. Afin de protéger ces données, des *rôles* sont associés à chaque catégorie de personne souhaitant accéder à celles-ci, et des ensembles de *règles* sont établis pour chacun de ces rôles. Ces règles peuvent explicitement définir qui devrait accéder à quelles données, mais également comment ces enregistrements devraient, ou *ne devraient pas*, être manipulés. Par exemple, un entraîneur pourrait être interdit d'accès aux données médicales *brutes*, mais pourrait quand même voir un *lissage* (c.-à-d., une moyenne) de celles-ci sur la semaine. Ici, le but du cycle de vie, à savoir l'ensemble des rôles et des règles associées, est d'assurer la confidentialité des données des athlètes et de leur permettre d'en garder le contrôle.

1.3.5 GESTION DES DROITS NUMÉRIQUES

La gestion des droits numériques (ou DRM, pour *Digital Rights Management*) peut être vue comme une forme spéciale de contrôle d'accès qui vise à contraindre la manière avec laquelle un document protégé par copyright peut être utilisé (Subramanya & Yi, 2006). Par exemple, dans le cas d'une image sujette aux droits d'auteurs, il peut être intéressant de limiter les modifications pouvant lui être apportées (*p. ex.*, recadrage, redimensionnement) et de spécifier quelles sont les actions que les utilisateurs sont autorisés à lui appliquer. Dans le cas extrême où aucune modification de l'image n'est autorisée, le seul cycle de vie valide serait le cycle de vie *vide*, exprimant que le document doit être laissé inchangé.

1.4 POINTS COMMUNS DES SCÉNARIOS

De manière plus ou moins évidente, tous les scénarios vu précédemment partagent deux points communs majeurs. Ceux-ci résident dans l'implication d'un ou plusieurs artefacts dans un processus défini, et dans la spécification, directe ou indirecte, de propriétés de cycle de

vie. La présente section a pour objectif de définir plus précisément les notions d'artefact et de cycle de vie, en s'appuyant notamment sur leur identification dans les scénarios précédents. En outre, le Tableau 1.2 résume l'ensemble des exemples vus en isolant, pour chacun d'entre eux, le type d'artefact considéré et les propriétés de cycle de vie que les manipulations successives sur celui-ci doivent respecter.

1.4.1 ARTEFACTS

La notion d'artefact est très générale. Dans le milieu des processus opérationnels, Nigam & Caswell (2003) sont les premiers à définir les artefacts comme des fragments d'information concrets, identifiables, et auto-descriptifs (c.-à-d., qui contiennent leurs données propres ainsi que des méta-données décrivant leur format et leur sens). Il est important de noter que la nature des données qu'ils contiennent ou leur représentation est sans intérêt : il n'y a aucune exigence sur le fait qu'un artefact soit plutôt un fichier texte, binaire, une base de données, ou même un document papier physique. De plus, la distinction d'artefacts repose uniquement sur leur identification : deux instances d'artefacts peuvent contenir exactement les mêmes données, au même format, mais peuvent rester distinctes.

De façon assez naturelle, dans notre premier scénario consacré à la gestion des dossiers médicaux, les artefacts sont représentés par les documents contenant les informations sur le patient, son traitement et les différents tests exécutés. Dans ce cas, l'artefact est un objet virtuel sous la forme d'un document informatique, mais ce dernier pourrait tout à fait être un document papier sans que ceci ne modifie quoi que ce soit au scénario.

Dans le deuxième scénario concernant la livraison de colis dans un contexte de logistique hyperconnectée, les artefacts sont symbolisés par les colis eux-mêmes. En l'occurrence, l'artefact désigne aussi bien l'objet physique que les informations du colis ; il établit donc une

Référence	Scénario	Artefacts	Cycles de vie
Bielova *et al.* (2009); Bielova & Massacci (2011)	Gestion de dossier médicaux	Dossier médical	Opérations permises
Section 1.1			Opérations permises et ordonnées, respectant l'intégrité et la confidentialité des données du patient
Section 1.2	Livraison de colis	Colis	Manipulations permettant un acheminement cohérent et sans irrégularité du colis
Mukkamala *et al.* (2008)	Processus bancaires	Dossier de demande de prêt	Ordonnancement d'actions autorisées visant à traiter une demande
Boebert & Kain (1985)	Politiques d'intégrité de données	Objets de données	Transformations assurant un transfert sécurisé des données
Brewer & Nash (1989); Sobel & Alves-Foss (1999)			Les manipulations d'objets par une entité ne peuvent résulter en un conflit d'intérêts
Biba (1977)			Écritures/lectures permises selon le niveau de fiabilité des objets
Section 1.3.3	Transports en commun	Carte à puce	Le nombre de trajets restants doit être suffisant et la séquence de trajets effectués cohérente
Johansen *et al.* (2015)	Sport de haut niveau	Données relatives aux performances sportives	Transformations permises des données pour en conserver la confidentialité
Subramanya & Yi (2006)	DRM	Œuvres sujettes aux droits d'auteurs	Modifications et utilisations permises de l'œuvre

TABLEAU 1.2 : Identification des notions d'artefacts et de cycle de vie dans les scénarios présentés © Quentin Betti.

correspondance entre la réalité physique (*p. ex.*, le colis est transporté d'un point à un autre) et sa représentation dans un système informatique (*p. ex.*, les coordonnées du colis sont mises à jour en fonction de sa position physique). Ce parallèle est rendu possible par l'attribution, dans le système, d'un identifiant au colis qui doit pouvoir lui être associé dans la réalité à l'aide, notamment, d'un code barre dans lequel est encodé son identifiant ou d'un tag RFID contenant celui-ci.

Les autres exemples identifient également des artefacts spécifiques. Dans les processus bancaires de Mukkamala *et al.* (2008), il s'agit d'un dossier de demande prêt qui, entre autres, est soumis à différentes approbations et nécessite la réalisation de paiements. Pour Boebert & Kain (1985), Brewer & Nash (1989), Sobel & Alves-Foss (1999) et Biba (1977), les artefacts considérés sont des objets de données manipulés par des diverses entités. Les cartes à puce des réseaux de transport sont elles aussi des artefacts physiques que les clients présentent aux différents lecteurs des stations pour que leur « crédit » de trajets soit vérifié ou mis à jour dans le système. Là encore, l'artefact est composé de données informatiques et d'un objet physique liés par un identifiant contenu dans la puce de la carte. Johansen *et al.* (2015), pour leur part, utilisent directement les données issues de performances sportives comme artefacts. Celles-ci sont créées par divers capteurs lors de l'observation du sportif en activité, puis consultées par les entraineurs, les préparateurs physiques et les médecins du sport. Enfin, dans le cadre du DRM, les artefacts considérés sont les œuvres sujettes aux droits d'auteurs.

1.4.2 CYCLES DE VIE

Toujours selon Nigam & Caswell (2003), un cycle de vie décrit le traitement d'un artefact dans son intégralité, de sa création à l'achèvement de son processus, ainsi que son archivage. Là encore, il s'agit d'une définition assez globale. Tel qu'utilisé dans ce projet, le terme cycle de vie fait référence à toutes les contraintes liées à l'artefact et à son traitement :

il définit quels sont les états que doit traverser l'artefact, quelles actions sont réalisables aux différentes étapes, qui est autorisé à les effectuer, et peut imposer des restrictions sur leur nature mais également sur leur contenu. Nous reviendrons en détail sur les contraintes que nous prenons en compte dans ce projet et leur formalisation.

Par exemple, dans notre premier scénario, le cycle de vie contraint les actions réalisables sur le dossier médical d'un patient. Ceci est fait par la définition de politiques de contrôle d'accès pour que seuls les pairs légitimes puissent manipuler ou consulter une donnée du dossier. De plus, ces actions sont également ordonnées pour que leur enchaînement corresponde à l'exécution cohérente du processus défini. Enfin, chaque manipulation de données doit respecter l'intégrité des données, pour que leurs formats et leurs valeurs restent, là aussi, cohérentes.

Le deuxième scénario reprend ces catégories de contraintes de cycle de vie pour qu'un colis puisse être livré rapidement à son destinataire, sans erreur ni irrégularité. Là aussi, les contraintes exprimées peuvent concerner l'ordonnancement des manipulations (*p. ex.*, il ne peut y avoir deux livraisons d'un colis sans ramassage intermédiaire), et certaines actions sont réservées à des acteurs spécifiques du processus (*p. ex.*, seul le client peut confirmer la réception du colis). Néanmoins, nous avons également vu qu'il était intéressant d'énoncer des contraintes d'intégrité faisant référence à des états passés de l'artefact (le colis). C'est notamment le cas quand nous déclarons que celui-ci doit toujours, à un seuil près, se rapprocher de sa destination finale, puisque ceci nécessite de comparer sa position actuelle à la précédente.

On peut également distinguer des contraintes de cycle de vie dans les autres exemples. Dans la demande de prêt bancaire de Mukkamala *et al.* (2008), les approbations et autres actions relatives à la demande doivent être effectuées dans un certain ordre et par des pairs spécifiques, tout en étant conditionnées par certaines variables propositionnelles. Dans le

cadre des politiques d'intégrité, Boebert & Kain (1985) stipulent que les données traversant un *assured pipeline* doivent subir certaines transformations spécifiques avant d'atteindre leur destinataire. Brewer & Nash (1989) et Sobel & Alves-Foss (1999), quant à eux, emploient le modèle du *Chinese Wall* pour empêcher les conflits d'intérêts ; ce modèle dicte qu'une entité qui manipule un objet ne puisse pas ensuite interagir avec d'autres objets partageant le même centre d'intérêt. La *low-water-mark ploicy* de Biba (1977) applique un type similaire de contraintes, en se basant cette fois sur une comparaison entre le niveau de fiabilité des données et celui du sujet. D'un autre côté, l'exemple des cartes à puce pour transports en commun énonce des contraintes de cycle de vie assez naturelles. Ainsi, seuls les détenteurs de cartes avec un nombre de trajets restants suffisant (en l'occurrence, il en faut au moins un) doivent être autorisés à embarquer. L'enchaînement des passages doit par ailleurs être cohérent : comme nous l'avons mentionné, deux passages successifs à la même station dans un court intervalle de temps pourrait caractériser une tentative de fraude, qu'il faut donc interdire. Le scénario des données sportives applique, quant à lui, des contraintes qui peuvent être mise au même plan que les *assured pipelines*. En effet, les données doivent subir des transformations précises avant d'être transmises, ce afin d'en assurer la confidentialité. Enfin, dans le cas du DRM, le cycle de vie spécifie directement les actions et les modifications réalisables sur les œuvres protégées, ce afin d'en contrôler l'utilisation.

CHAPITRE II
ÉTAT DE L'ART SUR LA SPÉCIFICATION ET L'APPLICATION DE CYCLES DE VIE

Dans le chapitre précédent, nous avons défini et illustré par quelques exemples les notions d'artefact et de cycle de vie. Dans ce chapitre, il s'agira donc d'exposer concrètement comment un cycle de vie peut être exprimé, puis *appliqué* (c.-à-d., comment on peut le faire respecter). Nous allons donc dresser un inventaire des travaux antérieurs portant sur la spécification et l'application des cycles de vie.

2.1 SPÉCIFICATION DES CYCLES DE VIE

Dans le milieu des processus opérationnels, les contraintes sur les artefacts ont été étudiées dans le contexte de « modèles de comportement d'objet ». La forme la plus courante de ce type de modèle est la modélisation par processus opérationnels centrés sur les artefacts (Nigam & Caswell, 2003; Bhattacharya *et al.*, 2007; Kumaran *et al.*, 2008; Hull *et al.*, 2011; Vaculín *et al.*, 2011). Dans ce contexte, divers artefacts peuvent être manipulés et transmis d'un pair à l'autre. Plutôt que (ou en plus) d'exprimer les contraintes en termes de ce que chaque pair peut faire, le processus opérationnel est défini par les cycles de vie des artefacts impliqués : toute séquence de manipulations qui est conforme au cycle de vie de chaque artefact est considérée comme une exécution valide.

La spécification de tels cycles de vie d'artefacts peut être réalisée par différents moyens que nous présentons ci-après.

2.1.1 MACHINES À ÉTATS FINIS

Les machines à états finis, ou automates finis, sont des modèles mathématiques permettant de caractériser l'évolution du comportement d'un système soumis à des influences extérieures, aussi appelées entrées. Ils sont utilisés notamment pour concevoir et vérifier du matériel électronique ou des composants logiciels, pour l'analyse grammaticale de langages (notamment lors de la compilation de programmes), ou encore pour rechercher des suites ou de caractères spécifiques au sein d'un long texte à travers les expressions régulières.

AUTOMATES FINIS

Concrètement, un automate fini est constitué d'un ensemble fini d'*états* et les entrées provoquent les *transitions* successives de l'état *courant* vers le prochain. Par souci de simplification, ici nous ne parlerons que d'automates finis *déterministes* qui, pour un état et une entrée, ont au plus un seul état destinataire (contrairement aux automates finis *non-déterministes* qui peuvent en avoir plusieurs). L'automate possède un état initial permettant de démarrer le processus, et il peut également avoir des états *acceptants* (aussi appelés finaux ou terminaux). Lorsque l'automate passe dans un état acceptant, on dit alors que l'entrée est *acceptée* par l'automate. Formellement, un automate fini peut être défini comme suit (Hopcroft *et al.*, 2006; Rosen, 2011; Gopalakrishnan, 2019).

Définition 1. *Un automate fini est un quintuplet* $\langle Q, \Sigma, \gamma, q_0, F \rangle$ *où :*

— Q *est un ensemble fini non vide d'états ;*

— Σ *est un alphabet fini non vide, c'est-à-dire l'ensemble des* symboles *(p. ex., un bit, une chaîne de caractères) en entrée de l'automate ;*

— $\gamma : Q \times \Sigma \to Q$ *est une* fonction de transition. *Pour un état et un symbole donnés, elle désigne l'état vers lequel l'automate doit transiter ;*

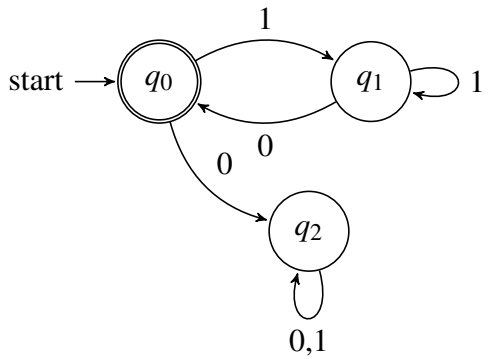


FIGURE 2.1 : Exemple d'automate fini © Quentin Betti.

— $q_0 \in Q$ *est l'état initial ;*

— $F \subseteq Q$ *est un ensemble d'états acceptants, potentiellement vide.*

Les diagrammes états-transitions sont les représentations graphiques des automates. Les états y sont représentés par des cercles ; l'état initial et les états finaux sont respectivement signalés par une flèche et des doubles cercles. Quant aux transitions, chacune d'entre elles est indiquée par un arc orienté entre deux états (il peut s'agir d'un même état) avec, pour étiquette, le symbole en entrée la déclenchant depuis l'état courant.

Pour illustrer ces notions, prenons l'exemple présenté en Figure 2.1. Ce diagramme montre un automate dont l'alphabet est $\{0, 1\}$, avec q_0 comme état à la fois initial et acceptant. Si le symbole en entrée est « 1 » on atteint l'état q_1, et on revient à q_0 uniquement si l'on rencontre un « 0 ». Si un « 0 » est fourni en entrée en q_0, on tombe dans l'état q_2. Depuis q_2, toutes les transitions ramènent systématiquement à celui-ci, que leur symbole associé soit « 0 » ou « 1 ». On qualifie souvent ce type d'état de « puits » ou « trou noir » car, une fois atteint, il est impossible d'en sortir, et ce quelque soit le symbole en entrée (Gopalakrishnan, 2019). Ceci permet de spécifier des automates qui invalident l'entrée dès lors qu'elle contient une certaine suite de symboles. Dans un premier temps, on peut déduire de cet automate qu'il

n'accepte que des entrées qui commencent par un « 1 » et finissent par un « 0 ». De plus, en observant les transitions depuis q_1, on s'aperçoit que l'état acceptant q_0 est atteint si, après une trame de « 1 » de taille indéterminée, un « 0 » est présent. Enfin, la transition de q_0 vers q_2 nous indique que dès lors qu'un « 0 » apparaît sans que le symbôle précemment consommé n'ait été un « 1 », alors l'entrée entière est invalidée (puisqu'on ne peut sortir de l'état q_2). On peut donc conclure que cet automate fini accepte uniquement les entrées vides ou les chaînes commençant par « 1 » et finissant par « 0 », contenant des trames de « 1 » séparées par exactement un « 0 ». Par exemple, les entrées « 10 », « 1010 », ou « 111110110111101110 » sont valides, à l'inverse de « 11 », « 01010 », ou « 1111100110 ».

Les automates finis peuvent être facilement adaptés aux domaines des processus opérationnels. En effet, il suffit pour cela de choisir comme alphabet l'ensemble des actions réalisables par les pairs sur l'artefact. De cette façon, il est possible de soumettre une séquence d'actions sur un artefact à l'automate correspondant à son cycle de vie. Si la séquence est acceptée par l'automate, alors on peut en déduire que le cycle de vie est respecté. Une partie des approches présentées dans cette section pour spécifier des cycles de vie sont d'ailleurs directement dérivées des automates.

TRANSDUCTEURS

Il faut néanmoins noter qu'une machine à états finis n'est pas restreinte seulement à l'acceptation ou non d'une entrée. Il existe en effet une famille de machines, appelés transducteurs, capables de produire des *sorties* lors du traitement de l'entrée. Par exemple, les machines de Moore (1956), n'acceptent ou n'invalident pas les entrées (c.-à-d., elles n'ont pas d'états acceptants). Plutôt, elles produisent des sorties déterminées par l'état courant de

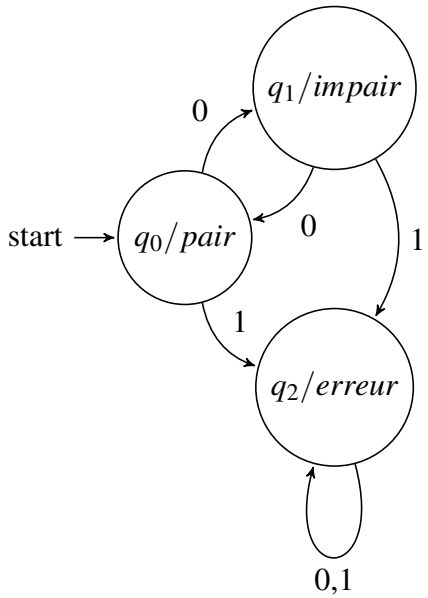


FIGURE 2.2 : Exemple de machine de Moore © Quentin Betti.

l'automate. La définition formelle des machines de Moore est similaire à celle des automates finis usuels (Rosen, 2011).

Définition 2. *Une machine de Moore peut être définie comme un sextuplet* $\langle Q, \Sigma, O, \gamma, g, q_0 \rangle$ *où :*

- Q *est un ensemble fini non vide d'états ;*
- Σ *est un alphabet d'entrée fini non vide ;*
- O *est un alphabet de sortie fini non vide ;*
- $\gamma : Q \times \Sigma \to Q$ *est une fonction de transition ;*
- $g : Q \to O$ *est une fonction de sortie, c'est-à-dire qu'elle assigne à chaque état un symbole produit en sortie ;*
- $q_0 \in Q$ *est l'état initial.*

Leur représentation est également très semblable, la seule différence étant que la sortie associée à un état est renseignée dans celui-ci via un séparateur (*p. ex.*, le caractère « ; » ou « / »). Prenons l'exemple d'un automate dont l'alphabet est $\{0,1\}$ et qui accepte seulement les entrées ne contenant aucun « 0 » et dont le nombre de « 1 » est impaire. On peut transposer cet automate en machine de Moore. Par définition, nous pouvons étendre la sortie de l'automate (auparavant limitée à l'acceptation ou non de l'entrée) à des résultats plus descriptifs. Ainsi, comme nous pouvons le voir dans la Figure 2.2, nous distinguons les cas où les entrées contiennent des « 1 », celles où le nombre de « 0 » est impair, et celles où ce dernier est pair. Ceci permet notamment une forme plus précise de contrôle basé sur une sortie plus descriptive de l'état de l'automate, là où seule une sortie binaire était possible auparavant.

La machine de Mealy (1955) est un autre de ces transducteurs, très similaire à la machine de Moore. En effet, la sortie des machines de Mealy, en plus de dépendre de l'état courant, est également déterminée par le symbole en entrée. Formellement, la définition d'une machine de Mealy est presque identique à celle de Moore (Rosen, 2011).

Définition 3. *Une machine de Mealy peut être définie comme un sextuplet* $\langle Q, \Sigma, O, \gamma, g, q_0 \rangle$ *où :*

- Q *est un ensemble fini non vide d'états ;*
- Σ *est un alphabet d'entrée fini non vide ;*
- O *est un alphabet de sortie fini non vide ;*
- $\gamma : Q \times \Sigma \to Q$ *est une fonction de transition ;*
- $g : Q \times \Sigma \to O$ *est une fonction de sortie qui assigne à chaque pair d'état et de symbole d'entrée un symbole de sortie ;*
- $q_0 \in Q$ *est l'état initial.*

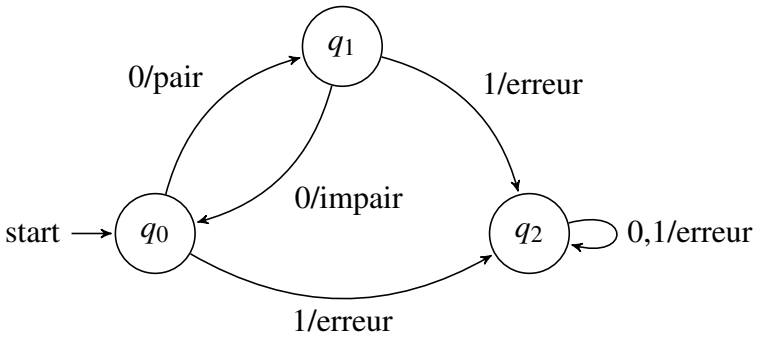


FIGURE 2.3 : Exemple de machine de Mealy © Quentin Betti

.

Là aussi, on peut transposer l'automate fini défini précédemment en machine de Mealy ; le résultat est présenté en Figure 2.3. Une sortie est cette fois-ci représentée sur chaque transition correspondante et se démarque du symbole en entrée associé par un séparateur (dans notre cas, le caractère « / »).

Il est important de noter que, malgré les similarités, il n'y a pas d'équivalence stricte entre machines de Moore et de Mealy. La différence essentielle vient de l'enchaînement des sorties : les machines de Moore produisent une sortie lorsqu'un état est atteint, tandis que celles de Mealy le font lorsqu'une transition est déclenchée. Ceci a comme conséquence que la sortie d'une machine de Moore est toujours plus longue d'un symbole par rapport à son « équivalent » en machine de Mealy. En effet, cette dernière n'émet pas de sortie lorsqu'elle se trouve à l'état initial (sans qu'il y ait eu de transition auparavant), contrairement à la machine de Moore.

2.1.2 RÉSEAUX DE PETRI

Introduits dans la thèse de doctorat de Carl Adam Petri (1962) et formalisés par la suite, les réseaux de Petri sont des graphes orientés qui permettent de décrire des processus par étapes (Petri & Reisig, 2008). Leur principal intérêt est qu'ils sont spécialement conçus pour

représenter des processus *concurrents*, là où les automates sont plus limités. Les réseaux de Petri se démarquent également de ces derniers par le fait qu'ils sont composés, non pas d'un, mais de deux types de nœuds différents reliés par des *arcs* : les états (communément appelés *places*) et les transitions. Les arcs sont orientés et chacun connecte un état vers une transition ou une transition vers un état.

DÉFINITION ET FONCTIONNEMENT

Le fonctionnement des réseaux de Petri se base sur le principe de consommation de *jetons*. Un ou plusieurs jetons sont initialement attribués à une ou plusieurs places du réseau et, lors de la prochaine étape, ceux-ci sont envoyés aux transitions correspondantes qui les *consomment* ; à l'étape d'après, les transitions *déclenchées* vont *produire* des jetons pour chaque état qui leur succède. Une transition peut avoir en entrée plusieurs arcs connectés à des états différents. Pour que la transition soit déclenchée, il faut alors que chacun des états qui la précèdent directement lui transmette un jeton ; à l'inverse, si une transition connectée à plusieurs états en sortie est déclenchée, alors elle produit un jeton pour chacun.

Pour être tout à fait précis, il est possible d'attribuer un *poids* à chaque arc, qui indique combien de jetons sont consommés (état vers transition) ou produits (transition vers état) lors de son parcours. Ainsi, les états peuvent contenir plusieurs jetons, jusqu'à une potentielle capacité maximale, et ces derniers peuvent être, par exemple, consommés tous en même temps par la prochaine transition ou à l'inverse distillés étape après étape si un seul jeton est requis. Formellement, un réseau de Petri peut donc être défini comme suit (Desel & Juhás, 2001) :

Définition 4. *Un réseau de Petri est un sextuplet* (S, T, F, M_0, W, K) *où :*

— S *est un ensemble non-vide d'états (ou de places) ;*

— T *est un ensemble non-vide de transitions ;*

— $F \subseteq (S \times T) \cup (T \times S)$ *est un ensemble non-vide d'arcs connectant un état vers une transition ou une transition vers un état ;*

— $M_0 : S \to \mathbb{N}$ *est le marquage initial, c.-à-d. pour chaque état le nombre de jetons initialement attribués ;*

— $W : F \to \mathbb{N}^+$ *attribue un poids à chaque arc ;*

— $K : S \to \mathbb{N}^+$ *attribue une capacité maximale à chaque état.*

REPRÉSENTATION GRAPHIQUE

Graphiquement, les réseaux de Petri sont représentés par des graphes où les états sont des cercles simples, les transitions des lignes noires épaisses (ou des carrés), et les jetons des petits ronds noirs à l'intérieur des états. Un exemple de réseau de Petri est donné en Figure 2.4. Ce dernier décrit le comportement d'une ampoule dont l'alimentation est commandée par un interrupteur.

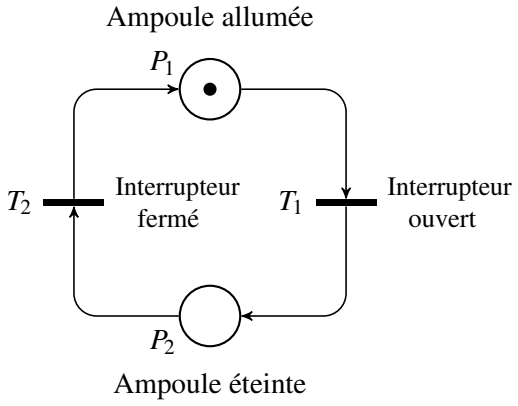


FIGURE 2.4 : Réseau de Petri caractérisant le fonctionnement simplifié d'une ampoule commandée par un interrupeur © Quentin Betti.

Trivialement, on observe deux états de l'ampoule : allumée (P_1) ou éteinte (P_2). Les positions de l'interrupteur, ouvertes ou fermées, sont représentées par les transitions T_1 et T_2,

et permettent respectivement de passer de l'état P_1 à P_2 et de P_2 à P_1. Les réseaux, comme celui de la Figure 2.4, dont les instants de déclenchement des transactions sont régis par des événements externes ou le temps sont qualifiés de *non autonomes* ; à l'opposé, les réseaux *autonomes* exposent des transitions donc les conditions de déclenchement ne sont ni explicites ni connus.

2.1.3 LOGIQUE TEMPORELLE LINÉAIRE

La logique temporelle linéaire (LTL), introduite par Pnueli (1977, 1979), fait partie de la famille des logiques *modales* temporelles. Ces dernières appliquent des *modalités* aux formules de logiques usuelles, telles que la logique de premier ordre ou la logique propositionnelle. Dans le cas de LTL, la modalité est celle du temps, comme son nom l'indique. Cependant, ce temps est abstrait et discret : il n'est pas mesurable, mais ordonne linéairement les états d'un système ou modèle. Concrètement, ceci revient à dire que, pour une séquence infinie d'états, il existe un unique état initial sans prédécesseur et que chaque état a un seul et unique successeur. LTL est un langage populaire pour exprimer des propriétés qu'un système réactif [6] doit respecter, telles que les propriétés de *vivacité* (*liveness properties*) de la forme « ceci peut se produire dans le système », et les propriétés de *sûreté* (*safety properties*) de la forme « ceci ne doit pas se produire dans le système ».

6. Les systèmes réactifs (Harel & Pnueli, 1985; Pnueli, 1986) sont des systèmes potentiellement composés de plusieurs processus fonctionnant en parallèle et en quasi-continu, le tout sous l'influence perpétuelle d'interactions et d'événements extérieurs. Ils sont opposés aux *systèmes transformationnels* qui se contentent d'accepter des entrées, puis de transiter d'état en état jusqu'à produire une sortie, mettant un point final au processus.

SYNTAXE

À l'origine, LTL se base uniquement sur des *variables propositionnelles*, qui sont des propositions atomiques dont la valeur est évaluée à « VRAI » ($\top$) ou « FAUX » ($\bot$). Par exemple, la proposition « le colis numéro 42 est expédié » est soit vraie, soit fausse. À l'aide des opérateurs logiques classiques $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction), $\rightarrow$ (implication) et $\equiv$ (équivalence), ainsi que celle des opérateurs temporels $\mathbf{X}$ et $\mathbf{U}$ (dont nous détaillons la signification plus tard), nous sommes capables d'énoncer des *formules* LTL selon les règles suivantes (Emerson, 1990) :

— toute proposition atomique est une formule LTL ;

— si φ et ψ sont des formules LTL, alors $\varphi \wedge \psi$ et $\neg\varphi$ sont des formules LTL ;

— si φ et ψ sont des formules LTL, alors $\mathbf{X}\,\psi$ et $\varphi\,\mathbf{U}\,\psi$ sont aussi des formules LTL.

L'opérateur $\mathbf{X}$ signifie « suivant » ; $\mathbf{X}\,\psi$ indique que la formule ψ est vraie dans l'état suivant du modèle. L'opérateur $\mathbf{U}$ représente « jusqu'à » ; $\varphi\,\mathbf{U}\,\psi$ signifie donc que ψ est vraie dans l'état courant ou dans un état futur et que φ doit être vraie jusqu'à cet état (à partir de ce dernier, φ n'a plus à tenir). À partir de ceux-ci, on peut définir d'autres opérateurs, notamment $\mathbf{G}$ (« globalement ») signifiant que la formule est tenue dans toute la séquence, et $\mathbf{F}$ (« dans le futur ») qui indique que la formule sera vraie à partir d'un état futur. Ce sont des abréviations directes des formules suivantes : $\mathbf{F}\,\varphi \equiv \top\,\mathbf{U}\,\varphi$ et $\mathbf{G}\,\varphi \equiv \neg\mathbf{F}\,\neg\varphi$.

SÉMANTIQUE

Supposons un système qui transite sur un ensemble infini d'états Q et qui, à chaque état $q_i \in Q$, génère un ensemble de propositions atomiques $s_i \in 2^{AP}$ où AP est l'ensemble des propositions atomiques ; la séquence résultante est alors $\bar{s} = (s_0, \ldots, s_i, s_{i+1}, \ldots)$. On écrit

$$
\begin{aligned}
\overline{s} \models p &\equiv p \in s_0 \\
\overline{s} \models \neg\varphi &\equiv \overline{s} \not\models \varphi \\
\overline{s} \models \varphi \wedge \psi &\equiv \overline{s} \models \varphi \text{ et } \overline{s} \models \psi \\
\overline{s} \models \mathbf{G}\,\varphi &\equiv \overline{s}^i \models \varphi \text{ pour tout } i \\
\overline{s} \models \mathbf{F}\,\varphi &\equiv \overline{s}^i \models \varphi \text{ pour un } i \\
\overline{s} \models \mathbf{X}\,\varphi &\equiv \overline{s}^1 \models \varphi \\
\overline{s} \models \varphi\,\mathbf{U}\,\psi &\equiv \overline{s} \models \psi, \text{ ou } \overline{s} \models \varphi \text{ et } \overline{s}^1 \models \varphi\,\mathbf{U}\,\psi
\end{aligned}
$$

FIGURE 2.5 : Sémantique formelle de LTL © Quentin Betti.

.

par $\overline{s} \models \varphi$ le fait que $\overline{s}$ satisfasse une formule LTL φ. Notons $\overline{s}^j$ le suffixe de $\overline{s}$ contenant son j-ième élément et les suivants. Soient $p \in AP$, φ et ψ des formules LTL, la sémantique formelle de LTL peut donc être résumée telle que dans la Figure 2.5. Depuis son introduction, cette sémantique a été étendue plusieurs fois, notamment pour être adaptée à la logique de premier ordre via LTL-FO$^+$ (Hallé & Villemaire, 2012; Khoury *et al.*, 2016).

EXEMPLE

Prenons l'exemple d'un système de commande et d'expédition de produits, où un client doit payer sa commande pour qu'elle lui soit expédiée. Nous émettons les propositions suivantes lorsque les quelques états suivants sont atteints :

— *paiementReçu* : émis lorsque le paiement de la commande est validé ;

— *expédié* : émis lorsque le colis contenant la commande a été expédié ;

— *enTransit* : émis lorsque le colis est en transit ;

— *livré* : émis lorsque le colis a été livré à sa destination finale.

Nous énonçons les deux propriétés suivantes :

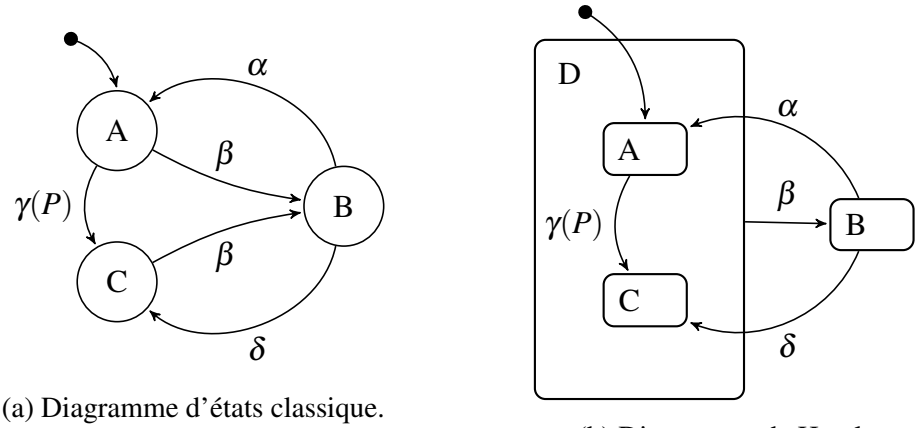
1. Le colis ne peut être expédié sans que le paiement correspondant à sa commande n'ait été validé au préalable ;
2. Tout colis expédié doit être livré, en passant potentiellement par un état de transit.

Une reformulation de la première propriété revient à dire qu'il ne peut y avoir de proposition *expédié* avant une proposition *paiementReçu*, ce qui se traduit par la formule : $\neg(\textit{expédié})\,\mathbf{U}\,\textit{paiementReçu}$. La seconde propriété indique qu'une fois qu'un colis a été expédié, il doit passer ou non par un état de transit avant d'être livré. Le fait qu'un colis soit ou non en transit avant d'être livré s'exprime naturellement avec la formule $\textit{enTransit}\,\mathbf{U}\,\textit{livré}$. Le reste de la formule revient à dire qu'il existe un instant où le colis est expédié et où il sera en transit ou livré dans l'instant d'après, ce qui donne finalement : $\mathbf{F}\,(\textit{expédié} \wedge \mathbf{X}\,(\textit{enTransit}\,\mathbf{U}\,\textit{livré}))$.

2.1.4 DIAGRAMMES DE HAREL

Pour des systèmes complexes, les machines à états et les réseaux de Petri présentent des possibilités de spécification limitées. En effet, pour un système soumis à des centaines de stimuli extérieurs le faisant transiter entre des milliers d'états différents, la représentation par un seul et même diagramme indivisible s'avère peu utilisable et maintenable sur le long terme. De plus, certaines mécaniques, comme la concurrence entre deux processus internes dans les diagrammes d'états, sont difficilement représentables de façon claire. C'est pour cette raison que, dans les années 1980 et 1990, de nombreux travaux sont menés par des développeurs ou des chercheurs afin de trouver le formalisme « idéal » pour décrire la conception et la spécification de systèmes complexes.

Dans cet objectif, David Harel (1987) étend le formalisme des automates finis et la représentation par diagrammes d'états des cycles de vie de processus, qu'ils soient informatiques ou issus d'autres domaines. Il propose donc une méthode de représentation adaptée à la



(a) Diagramme d'états classique.

(b) Diagramme de Harel.

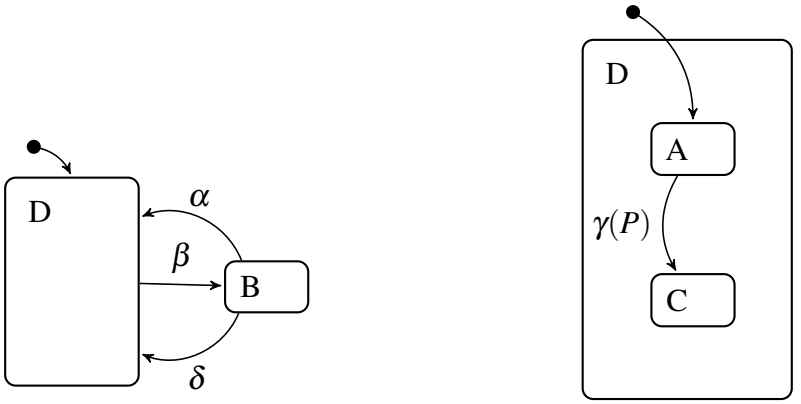
FIGURE 2.6 : Exemple de (a) diagramme d'états classique et (b) équivalent en diagramme de Harel avec super-état [7], tiré de Harel (1987). Reproduit avec la permission d'Elsevier.

spécification de systèmes réactifs. Cette proposition est motivée par le fait que la pertinence de la seule utilisation des machines à états est très limitée lorsqu'il s'agit de représenter des systèmes complexes, où le nombre d'états possibles peut évoluer de façon exponentielle et donner lieu à un diagramme peu structuré, difficilement maintenable ou utilisable. Les diagrammes de Harel, aussi appelés *statecharts*, fournissent donc des formalismes visuels étoffant ceux des diagrammes états-transitions classiques.

HIÉRARCHIE

La première notion ajoutée est celle de *hiérarchie* au sein des états en introduisant les *super-états* qui consistent en l'encapsulation d'états partageant des propriétés communes (*p. ex.*, transitions en commun, appartenance à une même sous-tâche du système), et les *sous-états* qui, à l'opposé, composent un super-état. Il est important de noter que ces propriétés

7. Dans ces deux diagrammes, la transition $\gamma(P)$ de l'état A vers C signifie ici que la condition P doit être vraie pour que la transition ait lieu. Cette notation fait partie des autres ajouts apportés par les diagrammes de Harel vus en Section 2.1.4.



(a) Abstraction des états A et C.

(b) Développement du super-état D.

FIGURE 2.7 : Abstraction de sous-états et développement de super-état dans le diagramme de Harel de la Figure 2.6b, tiré de Harel (1987). Reproduit avec la permission d'Elsevier.

ne sont pas exclusives : un super-état peut tout à fait être le sous-état d'un autre super-état, et inversement. Cet ajout permet de simplifier grandement les diagrammes, en réduisant notamment le nombre de transactions nécessaires. Prenons, par exemple, le diagramme d'états classique de la Figure 2.6a ; on peut y voir que les états A et C ont une transition avec un symbole identique vers l'état B. Dans un diagramme de Harel (Figure 2.6b), il est alors possible de regrouper l'état A et C dans un super-état D qui les englobe, et de remplacer la transition de A vers B et celle de C vers B par une seule et unique transition de D vers B. À la place de deux transitions, une seule est représentée sur le diagramme ; dans le cas de systèmes complexes, ce type de transformation peut donner lieu à la suppression de centaines de transitions superflues, et donc de simplifier grandement la lisibilité du diagramme.

ABSTRACTION ET DÉVELOPPEMENT

Ceci amène naturellement à une autre propriété des diagrammes de Harel qui est la possibilité d'*affiner* (*refine*) ou, au contraire, d'*abstraire* un ou plusieurs super-états. Comme on peut

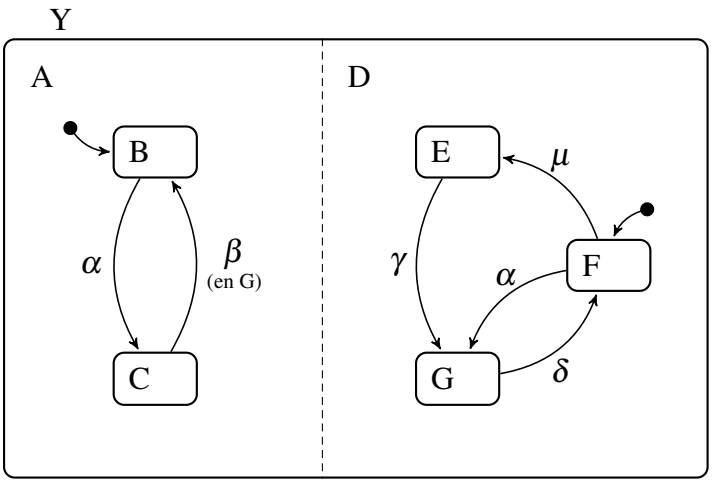


FIGURE 2.8 : Représentation de l'orthogonalité et de l'indépendance d'états dans un diagramme de Harel, tiré de Harel (1987). Reproduit avec la permission d'Elsevier.

le voir dans la Figure 2.7, il est possible d'effectuer un « dézoom » sur le diagramme précédent afin d'opacifier le contenu du super-état D (Figure 2.7a). Ceci permet de prendre un *cran de recul* sur la spécification et d'observer un comportement plus général du système. À l'inverse, il est possible de « zoomer » sur un super-état en particulier afin d'étudier son comportement interne (Figure 2.7b). Notons que, dans ces dernières figures, nous perdons l'information sur les transitions spécifiques de B vers A et C, telles qu'elles étaient représentées en Figure 2.6. Cependant, ce type de détails peut être retrouvé en augmentant ou en diminuant d'un cran (par rapport, respectivement, aux figures 2.7a et 2.7b) le zoom actuel sur le diagramme. Ces propriétés permettent, là aussi, de faciliter la conception et la lisibilité du diagramme en *isolant* davantage les ensembles d'états constituant, par exemple, une fonctionnalité précise du système sur laquelle on souhaite se concentrer à ce stade du processus.

ORTHOGONALITÉ

L'*orthogonalité* est une autre propriété majeure des diagrammes de Harel qui permet à deux états ou plus d'être actifs au même instant. Une telle relation est illustrée dans la Figure 2.8, où l'état Y est le *produit orthogonal* des super-états A et D. Quand on entre dans l'état Y, on est alors dans une combinaison d'états (B, F). Si l'événement α est soumis au système, alors on passe simultanément des états B à C et F à G, pour se retrouver dans l'état (C, G), ce qui exprime une forme de *synchronisation* entre A et D. Cependant, il existe également une *indépendance* entre ces derniers, puisque si la transition γ est déclenchée, D est affecté sans aucun impact sur A, transposant le diagramme de l'état (C, G) à (C, F). S'il y a indépendance, les deux super-états A et D sont néanmoins inter-connectés, puisque la condition « β (en G) » de l'état B à C indique que cette transition est déclenchée uniquement si le système est soumis à β et que l'état G est actif. L'orthogonalité permet de diminuer grandement le nombre de transitions et d'états qu'il serait nécessaire de représenter dans un diagramme classique, puisque celui-ci devrait alors inclure toutes les différentes combinaisons d'états possibles.

AUTRES PROPRIÉTÉS

Les diagrammes de Harel proposent également quelques ajouts mineurs, comme la possibilité de conditionner les transitions, tel qu'en Figure 2.6b où le déclenchement de la transition γ de A vers C est possible seulement dans le cas où la condition P est vraie. On note aussi le fait de pouvoir exprimer des temporisations (*p. ex.*, le système doit attendre deux secondes avant de passer d'un état à un autre) ou des délais maximums (*p. ex.*, le système ne peut rester dans cet état plus de 10 secondes), qui sont des conditions couramment rencontrées

dans les systèmes complexes. À la manière des machines de Mealy présentées plus tôt, il est également possible d'indiquer des « sorties » à produire lors du déclenchement de transitions. Ces sorties sont, comme pour les machines de Mealy, séparées d'un « / » avec le symbole de la transition, et sont généralement plutôt des actions entreprises par le système, tels que des symboles permettant de provoquer des transitions à un autre endroit du diagramme.

2.1.5 UML ET DIAGRAMMES DE COMPORTEMENT

De nos jours, le langage de modélisation graphique *Unified Modeling Language* ou UML (Object Management Group, 2017), dont la première version fut standardisée en 1997, est peut-être la solution la plus populaire (notamment dans le développement logiciel orienté objet) visant à standardiser la conception d'un système de façon visuelle. Pour ce faire, UML définit notamment l'usage et la notation d'un certain nombre de diagrammes, répartis en deux grandes catégories.

La première regroupe les diagrammes dits *de structure* décrivant les éléments *statiques* du système. On y trouve notamment les populaires diagrammes de classe qui, dans la programmation orientée objet, permettent de spécifier les différentes classes utilisées et leurs relations entre elles, ou encore les diagrammes d'objets qui décrivent les instances d'objets lors d'interactions spécifiques. La deuxième catégorie concerne les diagrammes *de comportement* qui, concrètement, indiquent les enchaînements d'actions et d'états que doit suivre le système. Cette dernière catégorie est donc particulièrement pertinente quant à la spécification des cycles de vie et nous allons détailler ici deux d'entre eux, à savoir les *diagrammes d'états-transitions* et les *diagrammes d'activité*. Ceux-ci s'appuient sur certains des formalismes et sémantiques introduits par les diagrammes de Harel et les réseaux de Petri, que nous avons présentés précédemment.

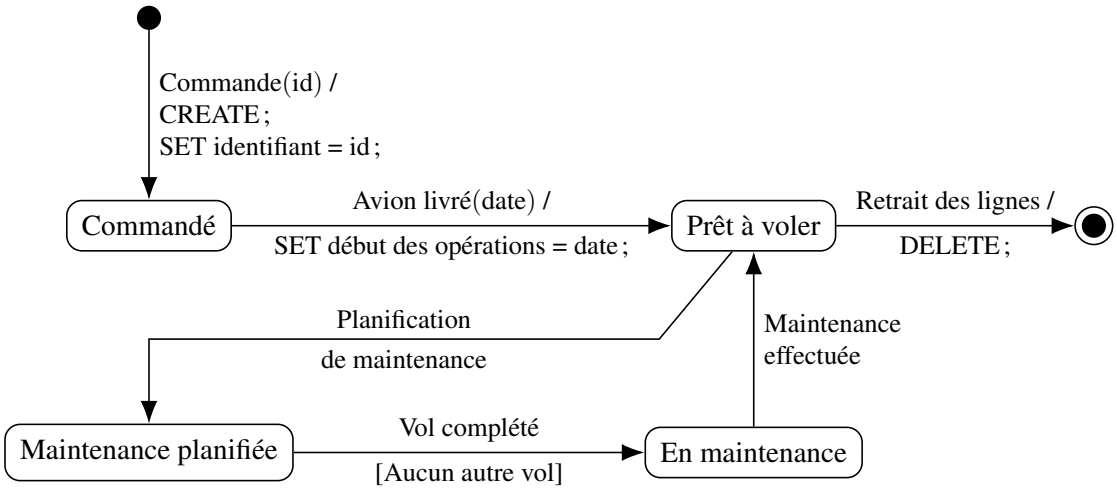


FIGURE 2.9 : Diagramme d'états UML simplifié de la gestion d'un avion par une compagnie aérienne © Quentin Betti.

DIAGRAMMES D'ÉTATS-TRANSITIONS

En UML, les diagrammes d'états-transitions (ou plus simplement, diagramme d'états) reprennent les principales notions des diagrammes de Harel pour spécifier les différents états possibles d'un système et ses transitions, tout en modifiant légèrement leur notation et en y ajoutant de nouveaux formalismes. Un exemple de diagramme d'états UML est donné en Figure 2.9 ; celui-ci est inspiré d'un scénario décrit par Baumann *et al.* (2005) et représente le cycle de vie simplifié de la gestion d'un avion par une compagnie aérienne, de sa commande au constructeur jusqu'à son retrait des lignes, en passant par ses périodes de maintenance.

La première différence avec les diagrammes de Harel est la présence de *pseudo-états* qui sont des états sans nom permettant d'indiquer certaines étapes du diagramme. Dans la présente figure, c'est le cas notamment de l'état initial (rond noir) et de l'état final (cercle contenant un rond noir). Aussi, on peut remarquer la syntaxe des transitions ; de manière générale, dans un

diagramme UML les transitions sont notées sous la forme (Lano, 2009) :

$$\textit{événement}(\textit{paramètres})[\textit{gardes}]/\textit{actions}.$$

L'*événement* est le symbole déclenchant la transition, accompagné potentiellement de *paramètres* ; il peut s'agir d'un événement provenant de l'extérieur ou du résultat d'une opération effectuée par le système. Ici, on peut noter la transition « Commande(id) », qui prend en paramètre l'identifiant de l'avion. Les gardes sont des conditions supplémentaires nécessaires à l'exécution de la transition ; par exemple, dans notre cas, pour qu'un avion dont la maintenance est planifiée soit effectivement envoyé en maintenance, il faut qu'il ait complété son vol actuel et qu'il n'ait plus aucun vol prévu. Enfin, les actions sont, comme dans les diagrammes de Harel et les machines de Mealy, des opérations à exécuter lors du déclenchement d'une transition, tel que dans la transition entre l'état initial et l'état « Avion commandé », où la commande de l'avion par la compagnie entraîne la création (action « CREATE ») de l'artefact correspondant dans le système et sa mise à jour (action « SET ») avec l'identifiant fourni en paramètre.

Il existe encore d'autres emprunts ou ajouts au formalisme des diagrammes de Harel, que nous ne détaillerons pas ici. On peut noter, par exemple, la possibilité de définir des régions orthogonales et donc de représenter la concurrence au sein d'un cycle de vie, de spécifier des variables d'état et d'effectuer des opérations sur celles-ci (*p. ex.*, incrémenter une variable jusqu'à atteindre une limite qui serait alors utilisée dans un garde), ou encore d'adopter le principe des machines de Moore, puisqu'il est possible de spécifier des actions à réaliser lors d'instants précis de l'activation d'un état, notamment à son entrée, pendant toute la durée de son activation, et à sa sortie.

DIAGRAMMES D'ACTIVITÉ

Un autre type de diagramme faisant particulièrement écho dans le monde des processus opérationnels est le diagramme d'activité qui représente le flux d'actions et de décisions nécessaires à l'accomplissement d'une tâche spécifique. Depuis UML 2.0, ces diagrammes s'inspirent de la représentation graphique des réseaux de Petri et y ajoutent quelques nouveautés.

Bien que l'on puisse retrouver l'aspect et le fonctionnement globaux des réseaux de Petri, on note tout de même un nombre important de différences. La première est d'ordre lexical puisqu'on y remplace les notions d'état et de transition, pour revenir à celle de *nœuds* d'activité. Ceux-ci regroupent plusieurs types :

— les *nœuds d'action* caractérisent le comportement désiré de l'activité. Ils décrivent l'action à exécuter par le système en accord avec son flux de jetons, et sont représentés par des rectangles aux bords ronds.

— les *nœuds d'objet* permettent de représenter les flots de données dans l'activité et sont indiqués par un rectangle aux bords carrés.

— les *nœuds de contrôle* spécifient la gestion du flux des jetons au sein de l'activité et sont composés de nœuds aux comportements bien définis. À l'inverse des réseaux de Petri, les jetons ne sont pas indiqués, mais leur trajet est implicitement représenté grâce aux nœuds et aux *arcs de contrôle*. Ces derniers, aussi appelés transitions (en rapport aux transitions des diagrammes d'états), sont des arcs orientés symbolisant le flux des jetons d'un nœud à un autre.

2.1.6 BPMN

BPMN (Object Management Group, 2011), pour *Business Process Model and Notation*, est un formalisme de représentation graphique de processus opérationnels, paru pour la première fois en 2004. Actuellement dans sa version 2.0, il adopte une approche très semblable aux diagrammes d'activité UML, en se basant sur des flux de tâches et de sous-processus influencés par des événements. Une différence notable, cependant, est que BPMN permet également la spécification de *flux de message* et donc la représentation des interactions entre plusieurs *participants* collaborant ou intervenant dans un même processus.

Comme UML, BPMN permet d'indiquer le flux de données et d'artefacts au sein d'un processus. Cependant, Meyer *et al.* (2013) montrent que ce type de représentation est très limitée notamment parce qu'elle reste abstraite sur la teneur des relations $m : n$ (relations *many-to-many*, c.-à-d. plusieurs objets d'un certain type peuvent être liés à plusieurs objets d'un autre type), si bien que le cycle de vie d'un artefact spécifique au sein d'un processus est difficilement identifiable. Dans ce contexte, Meyer *et al.* (2013) étendent BPMN avec une notation semblable aux bases de données relationnelles en assignant, notamment, des clés primaires et étrangères aux objets créés ou manipulés dans les diagrammes d'activité. Lorsque plusieurs instances d'objet sont attendues dans un processus, ceci permet de les identifier et d'énoncer des conditions qui leur sont propres, rendant possible la mise en vigueur potentiellement automatique de leur cycle de vie.

2.1.7 BEDL

De façon plus générale, les outils de modélisation de processus opérationnels ont tendance à se concentrer davantage sur le flux des tâches et activités et délaissent la gestion des données qu'elles manipulent. Ces dernières sont souvent représentées par des variables

de processus de façon implicites ou, comme nous l'avons vu dans la section précédente, par des objets dont le comportement et les relations avec le processus ou d'autres objets sont peu définis. Dans ce contexte, Nandi *et al.* (2010) introduisent le *Business Entity Definition Language*, un langage de spécification au format XML mis au point pour décrire l'évolution de *Business Entities* (BEs), des objets conceptuels constitués des quatre composants suivants.

Tout d'abord, les BEs possèdent un *modèle d'information*. Celui-ci, comme son nom l'indique, contient les informations sur les artefacts du processus sous la forme d'un ensemble de paires attributs-valeurs. Par exemple, dans le cas d'un processus de livraison de colis (Nandi *et al.*, 2010), il peut s'agir entre autres des informations sur l'expéditeur, le destinataire, le colis lui-même ou son transit (*p. ex.*, jour et heure d'expédition/livraison à un centre de tri ou à la destination finale).

Deuxièmement, on assigne aux BEs un cycle de vie qui, comme nous l'avons vu dans les sections précédentes, décrit la ou les séquences de tâches pouvant être réalisées sur ces objets. Ce cycle de vie est spécifié sous la forme d'un automate fini et exprimée en BEDL via des balises XML. L'automate est indiqué par une balise `<lifecycle>` dans laquelle on déclare les états (dont l'initial) dans une balise `<states>` et les transitions sous la forme d'éléments `<transition/>` au sein de la balise `<transitions>`.

Viennent ensuite les *politiques d'accès*, qui définissent les droits des différents *rôles* présents dans le processus à réaliser certaines opérations sur un modèle informationnel d'artefact. Celles-ci sont basées sur le modèle *CRUD* (pour *Create, Read, Update, Delete*) spécifique au rôle de l'intervenant et à l'état courant du processus. Ceci signifie que, pour chaque rôle et à un état du cycle de vie donné, il est possible ou non de créer, lire, mettre à jour ou supprimer les informations indiquées.

Enfin, les *notifications* permettent d'émettre des événements à partir des actions de type CRUD réalisées ou des transitions du cycle de vie qui ont été déclenchées. Des intervenants extérieurs, tels que d'autres processus, peuvent ainsi écouter ces événements et agir en fonction si nécessaire.

2.1.8 CONTRAINTES DYNAMIQUES D'ARTEFACTS RELATIONNELS

Dans le même objectif d'adapter la spécification de cycle de vie à des modèles d'information d'artefact complexes, Hariri *et al.* (2011) présente le langage formelle μL, une variante du μ-calcul, pour exprimer des contraintes internes aux artefacts utilisant des modèles de bases de données relationnelles comme modèle d'information. Ils définissent la notion d'*artefact relationnel* comme suit :

Définition 5. *Un artefact relationnel est un triplet* $A = \langle \boldsymbol{R}, I_0, \Phi \rangle$ *où :*

- $\boldsymbol{R} = R_1, \ldots, R_n$ *est un schéma de base de données, c.-à-d. un ensemble de schémas de relations ;*
- I_0 *est une instance de base de données correspondant au schéma* $\boldsymbol{R}$ *et représente l'état initial de l'artefact ;*
- Φ *est une formule* μL *portant sur* $\boldsymbol{R}$ *constituée par la conjonction de toutes les contraintes dynamiques intra-artefact de A. Autrement dit, satisfaire* Φ *revient à satisfaire la totalité des contraintes exprimées pour les composants de* $\boldsymbol{R}$*, c.-à-d.* $R_1, \ldots, R_n$.

Les auteurs illustrent cette définition par un exemple de processus d'approvisionnement au sein d'une entreprise. Lorsqu'un employé interne à une entreprise a besoin d'un produit spécifique, celui-ci, alors appelé *demandeur*, doit en faire la *requête* à un autre employé, l'*acheteur*, qui va ensuite devoir *s'approvisionner* auprès de *fournisseurs* externes. On peut décomposer le système en deux artefacts relationnels `ReqOrders` et `ProcOrders`, qui

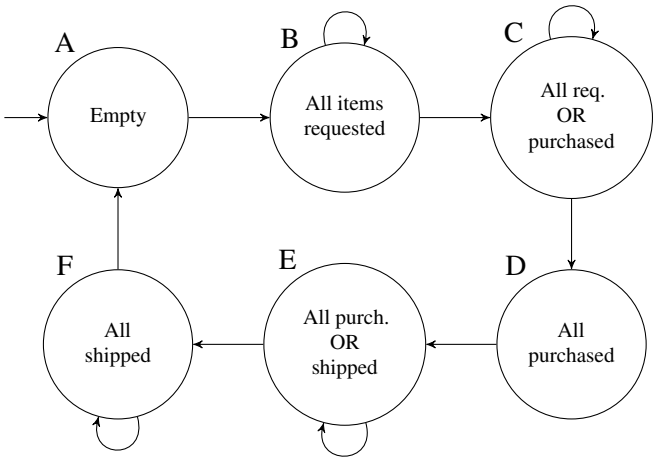
contiennent la totalité des données sur les requêtes et les demandes d'approvisionnement, respectivement. `ReqOrders` est défini par le triplet $\langle R_{RO}, I_{0,RO}, \Phi_{RO} \rangle$ où (nous simplifions ici l'exemple original) :

— R_{RO} = ROItem(*RoCode, ProdName, Status*). ROItem est une relation caractérisant les produits faisant l'objet d'une ou plusieurs requêtes. Elle est composée des attributs *RoCode* (identifiant de la requête), *ProdName* (nom du produit demandé), et *Status* (statut du produit, c.-à-d. demandé, acheté, ou expédié) ;

— $I_{0,RO}$, conformément à la définition précédente, est l'état initial de R_{RO}, qui est donc celui de son unique composant ROItem. À l'état initial la relation ROItem doit être vide.

— Φ_{RO}, qui peut être représentée par la Figure 2.10.

La Figure 2.10a est une représentation informelle de Φ_{RO}, où le processus est représenté par un automate dans lequel chaque état contient sa propre contrainte. Les formules μL correspondant à ces contraintes sont exprimées pour chacun des états A à F dans la Figure 2.10b. Ainsi, la formule ψ_A, qui caractérise l'état initial A, signifie que la relation ROItem doit être vide. Ensuite, à l'état B, des produits sont requis par le demandeur ; ψ_B indique donc que tout produit dans la relation doit avoir le statut « demandé ». Puis, il va exister un état où certains produits seront achetés par l'acheteur, et d'autres pas encore ; il faut donc que, pour tout produit, son statut soit « demandé » ou « acheté ». Et ainsi de suite, jusqu'à l'état F, où ψ_F énonce que tous les produits doivent avoir été expédiés par les fournisseurs. Finalement, lorsque tous les produits ont été reçus, on revient à l'état A en supprimant toutes les entrées de ROItem.



(a) Description informelle.

$$
\begin{aligned}
\psi_A &= \neg\exists x,y,z.ROItem(x,y,z) \\
\psi_B &= \forall x,y,z.(ROItem(x,y,z) \rightarrow z = requested) \\
&\quad \wedge \exists x,y.ROItem(x,y,requested) \\
\psi_C &= \forall x,y,z.(ROItem(x,y,z) \rightarrow (z = requested \vee z = purchased)) \\
&\quad \wedge \exists x,y.ROItem(x,y,requested) \wedge \exists x,y.ROItem(x,y,purchased) \\
\psi_D &= \forall x,y,z.(ROItem(x,y,z) \rightarrow z = purchased) \\
&\quad \wedge \exists x,y.ROItem(x,y,purchased) \\
\psi_E &= \forall x,y,z.(ROItem(x,y,z) \rightarrow (z = purchased \vee z = shipped)) \wedge \\
&\quad \exists x,y.ROItem(x,y,purchased) \wedge \exists x,y.ROItem(x,y,shipped) \\
\psi_F &= \forall x,y,z.(ROItem(x,y,z) \rightarrow z = shipped) \wedge \exists x,y.ROItem(x,y,shipped)
\end{aligned}
$$

(b) Notation formelle.

FIGURE 2.10 : Une contrainte intra-artefact dynamique (a) et sa formalisation en μ-calcul (b) ; adapté de Hariri *et al.* (2011). Reproduit avec la permission de Springer.

2.1.9 PARADIGME GUARD-STAGE-MILESTONE

Un autre moyen de modéliser le cycle de vie d'artefacts est le paradigme *Guard-Stage-Milestone* (GSM) introduit par Hull *et al.* (2011). Cette approche identifie quatre éléments clés : un modèle d'information pour les artefacts ; des *jalons* (*milestones*) qui correspondent à des objectifs opérationnels ; des *phases* (*stages*), qui correspondent à des clusters d'activités visant à accomplir les jalons ; et finalement des *gardes*, qui contrôlent l'activation des phases. Les jalons et les gardes sont contrôlés de manière déclarative, basée sur le déclenchement d'événements et/ou de conditions.

2.2 APPLICATION DE CYCLES DE VIE

Appliquer un cycle de vie revient à s'assurer que toutes les exécutions potentielles d'un système respecte sa spécification ; le fait qu'un système ait un comportement *correct* est donc relatif à celle-ci. Comme nous l'avons vu dans la section précédente, la spécification de cycles de vie d'artefact est un domaine relativement bien étudié. Ici, nous présentons certains des nombreux travaux portant sur l'application de cycles de vie, que nous répartissons en trois catégories : les approches *centralisées*, celles par *vérification statique*, les *décentralisées*. Il nous semble néanmoins essentiel d'introduire en premier lieu les principales méthodes formelles utilisées pour la vérification logicielle.

2.2.1 PRÉLIMINAIRES SUR LES MÉTHODES DE VÉRIFICATION

Dans cette section, il s'agira de définir les notions fondamentales de la vérification logicielle, à savoir le *model checking* et le *runtime verification*. En effet, la plupart des travaux cités dans les sections suivantes reprennent ces principes, d'où l'importance de les introduire ici. Nous exposons également le *Complex Event Processing* (CEP) qui, même s'il

n'est pas particulièrement restreint à la vérification de cycles de vie, n'en reste pas moins pertinent lorsque combiné aux méthodes formelles usuelles. Dans ce contexte, nous présentons également un moniteur pour CEP, BeepBeep 3 (Hallé, 2018), utilisé dans certains des travaux mentionnés ainsi que dans le Chapitre 4 de cette thèse pour spécifier *et* monitorer des propriétés de cycles de vie.

VÉRIFICATION DE MODÈLES

La vérification de modèles, plus couramment appelée *model checking*, repose sur le principe d'abstraction du système étudié en un *modèle* (Clarke *et al.*, 2018). Le système est ainsi représenté par celui-ci, souvent sous la forme d'une structure de Kripke (1959), c.-à-d. un graphe orienté où les arcs sont des transitions et les nœuds des états étiquetés par des ensembles de variables propositionnelles atomiques. Le modèle expose les différentes successions d'états possibles, aussi appelées *chemins* ; en accord avec l'étiquetage des états, un chemin résulte donc en une séquence d'ensembles de variables propositionnelles.

En représentant le système par une structure de Kripke, le but est donc de définir les propriétés comme des séquences d'ensemble de variables propositionnelles possibles. La logique temporelle est particulièrement adaptée à l'expression de telles propriétés ; nous l'avons vu avec LTL dans la Section 2.1.3, mais d'autres langages comme CTL, pour *Computational Tree Logic* (Clarke & Emerson, 1981), ou CTL* (Emerson & Halpern, 1986) sont tout aussi utilisables.

Grâce à ce type de modélisation et d'expression de cycle de vie, il est donc possible d'effectuer une vérification *exhaustive* du comportement du système. Cependant, en pratique, il peut s'avérer complexe de modéliser un système réel de la sorte. De plus, nous le verrons, la vérification d'un modèle peut représenter un véritable défi.

VÉRIFICATION EN TEMPS-RÉEL

La vérification en temps-réel (*p. ex.*, Bartocci *et al.*, 2018), plus connue sous le nom de *runtime verification* ou *runtime monitoring*, propose une philosophie de vérification diamétralement opposée à celle du model checking. Dans ce dernier, la vérification s'effectue sur un modèle abstrait du système, *a priori* d'une quelconque exécution de celui-ci ; à l'inverse, le runtime verification s'appuie uniquement sur les résultats de l'exécution en temps réel du système.

En effet, le système considéré est amené à produire divers événements informant sur, par exemple, son état interne ou ses opérations actuelles. Une telle séquence d'événements est communément appelée une *trace*. Au fur et à mesure que cette trace se crée, un *moniteur* est chargé d'observer celle-ci afin d'évaluer les propriétés de cycle de vie. Là encore, de telles propriétés peuvent être exprimées grâce aux formalisations vues en Section 2.1.

Les exécutions *correctes* sont donc identifiées par les séquences d'événements produites par le système. Le runtime verification ne permet donc pas rigoureusement de prouver que le système est *valide*, car un sous-ensemble d'exécutions correctes parmi l'ensemble, potentiellement infini, des exécutions possibles du système ne permet pas de généraliser la validité du comportement global. Plutôt, le runtime verification permet de mettre en évidence les erreurs ou bogues à travers l'analyse des traces ; ceci le différencie totalement du model checking qui, lui, vise à englober les comportements possibles du système à travers un modèle. Cependant, on constate qu'il est en pratique bien plus facile d'utiliser le runtime verification, puisqu'il suffit d'effectuer un certain nombre d'exécutions du système et d'y appliquer les propriétés, là où le model checking nécessite la conception et vérification d'un modèle suffisamment représentatif.

LE COMPLEX EVENT PROCESSING

Les flux d'événements représentent une part importante de la masse de données produits par les systèmes informatiques. Ils peuvent être générés par une myriade de sources tels que les capteurs (Wang *et al.*, 2013; Jia *et al.*, 2009; Hallé *et al.*, 2016a), les journaux de processus opérationnels (van der Aalst, 2016), l'analyse de code informatique (Calvar *et al.*, 2012; Leucker & Schallhart, 2009; Jin *et al.*, 2012), les transactions financières (Adi *et al.*, 2006), les systèmes de santé (Berry & Milosevic, 2013), la capture de paquets réseau (La *et al.*, 2016), ou, comme c'est le cas dans le Chapitre 4, par les transactions d'une blockchain. La capacité à collecter et traiter ces données peut être mise à bon escient dans de nombreux domaines tels que le test logiciel, le forage de données, ou la vérification de conformité.

Le *Complex Event Processing* (CEP) peut être brièvement résumé à l'analyse et l'agrégation de données produites par des systèmes d'information émettant des événements (Luckham, 2008). Une principale fonctionnalité du CEP est la possibilité de corréler les événements venant de sources multiples, ayant lieu à divers instants. Les informations extraites de ces événements sont traitées, et peuvent mener à la création de nouveaux événements « complexes », composés de ces données précédemment calculées. Ce flux d'événements complexes peut ensuite être lui-même utilisé en tant que source d'un nouveau traitement, ainsi qu'agrégé et mis en corrélation avec d'autres événements.

Le traitement d'événements peut s'effectuer en deux modes d'opération distincts. Dans le mode en ligne (ou « streaming »), les événements sources sont consommés en même temps qu'ils sont produits, les événements en sortie sont progressivement calculés et mis à disposition. Le flux de sortie est généralement considéré comme monotone : une fois qu'un événement est produit en sortie, il ne peut être « repris » plus tard. Au contraire, dans le mode hors ligne (ou « batch »), le contenu du flux source est complètement connu à l'avance (en étant stocké sur un

disque ou dans une base de données par exemple). Parfois, le fait qu'un système fonctionne en ligne ou hors ligne peut importer : par exemple, les traitements hors ligne ont l'avantage de pouvoir indexer, rembobiner ou accélérer le flux d'entrée sur demande.

Dans le contexte de la vérification, il est donc possible de traiter, en temps réel ou *a posteriori*, des événements provenant de systèmes ou composants logiciels potentiellement multiples, puis d'évaluer des propriétés sur ceux-ci par l'application d'opérations complexes représentant, en quelque sorte, la spécification du cycle de vie à respecter. L'intérêt est double puisque, comme nous le verrons au Chapitre 4, il est possible de réaliser, potentiellement en simultané, la vérification de propriétés et l'extraction de données analytiques du système en utilisant le même outil.

BEEPBEEP : UN MONITEUR POUR CEP

BeepBeep 3 (Hallé, 2018) est un moteur de traitement de flux d'événements implémenté sous la forme d'une bibliothèque Java open source [8]. Il est organisé autour du concept de *processeurs*. Pour résumer, un processeur est une unité basique de calcul qui reçoit une ou plusieurs traces d'événements en entrée, et produit une ou plusieurs traces en sortie. La bibliothèque BeepBeep de base fournit une poignée de processeurs génériques réalisant des opérations élémentaires sur les traces, et sont résumés dans la Figure 2.11 ; ils peuvent être représentés par des boîtes avec des « tuyaux » d'entrée/sortie.

Les principaux processeurs fournis par la bibliothèque de base sont décrits comme suit. Étant donnée une fonction f, le processeur `ApplyFunction` applique f à chaque événement d'entrée e et retourne $f(e)$ en sortie. Une fonction peut avoir de multiple arguments en entrée ;

8. https://liflab.github.io/beepbeep-3

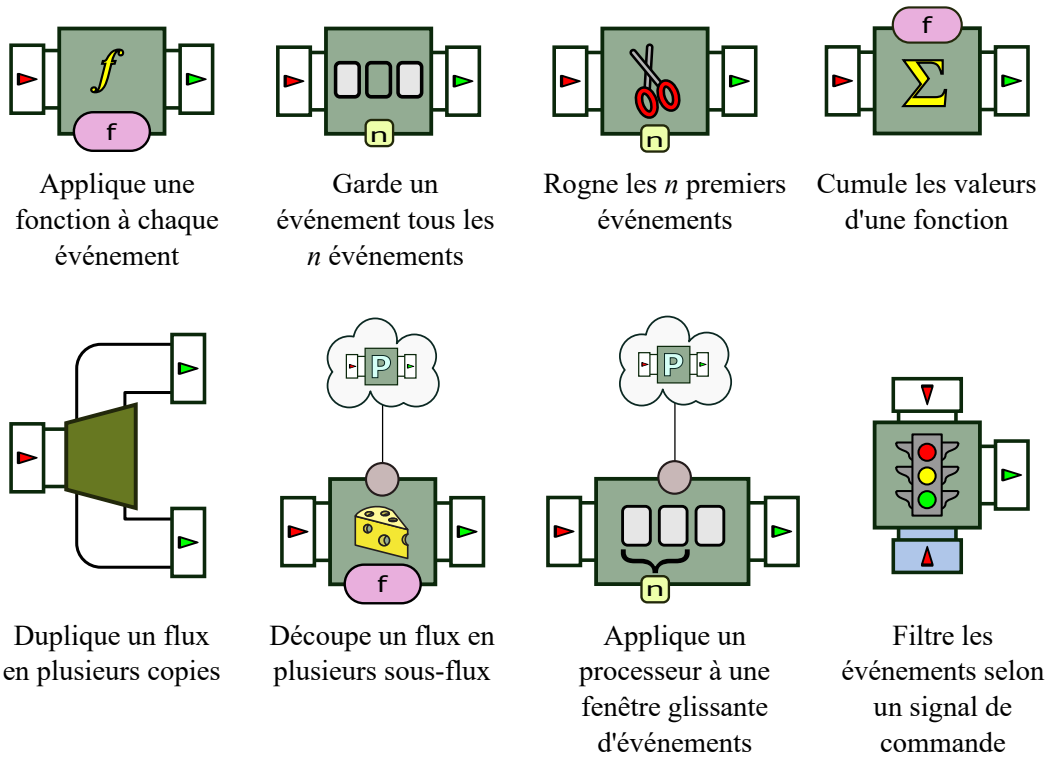


FIGURE 2.11 : Les processeurs basiques de BeepBeep 3 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

dans ce cas, le processeur correspondant a un nombre de tuyaux d'entrée égal à l'arité de la fonction. Le processeur `Cumulate` calcule une « agrégation » cumulative sur les événements reçus jusqu'à présent. Une fonction pouvant être définie par l'utilisateur f est appliquée au flux ; cette fonction doit avoir deux arguments. Si l'on note a la valeur précédemment retournée par le processeur et e le nouvel événement, le résultat en sortie sera $f(a,e)$. Une valeur initiale a' doit être définie pour produire le premier événement. Le rôle du processeur `Trim` est de supprimer un nombre fixe d'événements depuis le début du flux. Le nombre d'événements à supprimer est spécifié en argument lorsque le processeur est instancié. Réciproquement, le processeur `Pad` insère un événement un nombre prédéfini de fois, avant de renvoyer les événements produits par un autre processeur. Le processeur `Decimate` rejette les événements d'un flux d'entrée selon des intervalles périodiques. Cette tâche peut être réalisée de deux façons : sur la base d'un nombre d'événements fixe (`CountDecimate`), ou d'un intervalle de temps fixe (`TimeDecimate`).

Le principe du processeur `Fork` est de diviser le flux originel en plusieurs copies identiques. Cette division permet d'appliquer séparément différents calculs sur le même flux. Le processeur `Filter` fournit à l'utilisateur la possibilité de garder ou supprimer arbitrairement les événements d'un flux. Dans sa forme basique, un `Filter` a deux entrées et une sortie. Le premier tuyau reçoit le flux d'événements à filtrer et le second des booléens. Si la valeur du booléen à la position n est $\top$ (vrai), l'événement à la position n du flux est bien envoyé en sortie.

Comme son nom l'indique, le processeur `Window` réalise une opération sur une fenêtre coulissante d'événements du flux d'entrée. Une fenêtre de n événements successifs est accumulée, puis alimente un autre processeur P. Le résultat de P sur cette fenêtre représente l'événement de sortie de la fenêtre. L'arrivée de nouveaux événements pousse les anciens en dehors de la fenêtre, et le calcul recommence avec la nouvelle fenêtre. Une fonctionnalité

intéressante de BeepBeep est que P n'est pas restreint à de classiques agrégations telles que des sommes ou des moyennes ; n'importe quel processeur peut être utilisé sur un calcul de fenêtre, traitements non-numériques inclus.

Enfin, le processeur `Slice` est un des plus complexes de la bibliothèque de base. Il utilise une fonction f pour séparer un flux d'événements en plusieurs sous-flux. Chacun de ces sous-flux est ensuite envoyé à différentes instances du même processeur P, et le résultat de chaque copie est agrégé dans un tableau associatif. Par exemple, un processeur `Slice` pourrait être utilisé pour séparer un journal de processus opérationnel en différentes instances de processus (basées sur un identifiant), et réaliser un traitement P sur chaque instance séparément.

Dans le but de créer des traitements personnalisés sur les traces d'événements, BeepBeep permet aux processeurs d'être composés ; cela signifie que chaque sortie d'un processeur peut être redirigée vers l'entrée d'un autre, créant des chaînes de processeurs complexes. Les événements peuvent être « poussés » vers les entrées de la chaîne, ou « tirés » depuis ses sorties, et BeepBeep est responsable de la gestion implicite des files d'événements associées aux entrées et sorties de chaque processeur. De plus, un utilisateur peut également créer ses propres processeurs et fonctions, en héritant respectivement des classes `Processor` et `Function`. Les extensions de BeepBeep avec des objets prédéfinis personnalisés sont appelées *palettes* ; il existe des palettes à des fins variées, telles que le traitement de signal, la manipulation XML, le traçage de graphiques, et l'implémentation d'automates finis.

Ces dernières années, BeepBeep a été impliqué dans de nombreux cas d'étude (Hallé & Villemaire, 2012; Hallé *et al.*, 2016a,b,c; Khoury *et al.*, 2016; Boussaha *et al.*, 2017; Varvaressos *et al.*, 2017; Roudjane *et al.*, 2018), et permet nativement la spécification de langages dédiés (Hallé & Khoury, 2018).

2.2.2 APPROCHES CENTRALISÉES

Nombreux sont les travaux se basant sur le fait que les artefacts seront manipulés à travers un moteur de gestion des processus (*workflow engine*). De fait, les fonctionnalités requises pour appliquer les contraintes du cycle de vie peuvent être implémentées directement à son emplacement central, puisque tous les accès en lecture ou écriture aux documents seront faits à travers le système.

C'est le cas, par exemple, des travaux faits par Zhao *et al.* (2009) et Gerede & Su (2007). Ceux-ci considèrent les artefacts d'un processus comme des objets classiques de programmation orientée objet, à ceci près que chacun est caractérisé par un automate fini, en plus des attributs et méthodes usuels ; les transitions de cet automate sont étiquetées par les noms des méthodes de l'objet. Ceci leur permet, entre autres, de définir des contraintes TiLE, pour *Time Line Expression* (Zhao *et al.*, 2009), et ABSL, pour *Artifact Behavior Specification Language* (Gerede & Su, 2007), qui s'appliquent à des *instantanés* du système, c.-à-d. à des *configurations* de tous les états et attributs du système pour des instants donnés. Assurer le respect du cycle de vie du système en temps réel revient alors à vérifier que, pour chaque nouvelle action entreprise, la configuration résultante respecte chacune des contraintes exprimées. Si ce n'est pas le cas, les actions sont tout bonnement bloquées.

Un travail similaire a été réalisé au niveau des bases de données (BDD) : Ataullah & Tompa (2011) proposent une technique pour convertir des contraintes exprimées par des automates finis en déclencheurs de système de gestion de BDD (SGBD). Dans cet article, les artefacts traités existent sous la forme d'entrées de relation de BDD. Un état de l'artefact correspond donc à une configuration spécifique de ses attributs et d'informations dérivées de ceux-ci. Par exemple, on peut définir un état *NewAndUnpaid* (voir Figure 2.12) pour des factures impayées qui ont été créées récemment, tel que la condition associée

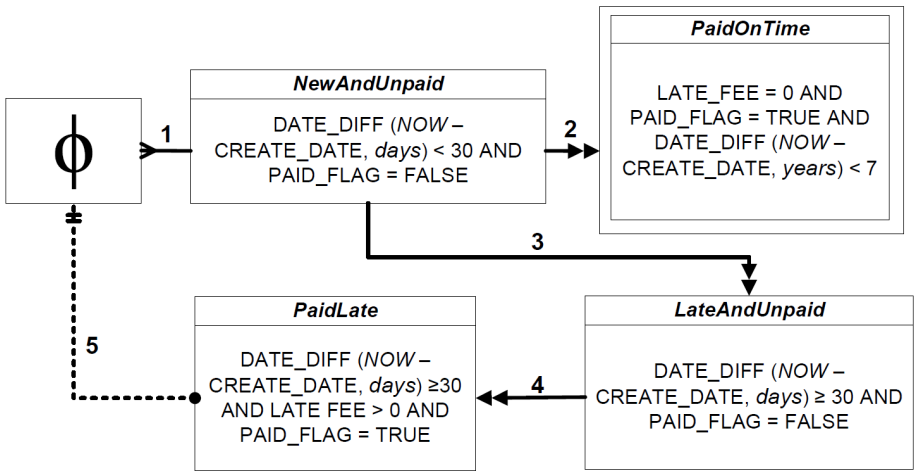


FIGURE 2.12 : Représentation d'un exemple de cycle de vie reposant sur les transitions possibles entre les états d'une entrée de base de données correspondant à une facture, de Ataullah & Tompa (2011). Image Creative Commons.

« DATE_DIFF(NOW − CREATE_DATE) < 30 AND PAID_FLAG = false » soit vérifiée. En outre, les auteurs définissent une représentation graphique de ces états qui permet de spécifier les transitions possibles entre ceux-ci. Par exemple, dans la Figure 2.12, la transition #1 signifie qu'il suffit que la condition de l'état *NewAndUnpaid* soit respectée pour que la transition ait lieu (l'état ϕ étant un équivalent d'état initial), tandis que les transitions #2, #3 et #4 impliquent que l'état qui précède chaque transition ait eu lieu *et* que la condition de l'état cible soit respectée pour activer la transition. Le projet est donc basé sur un modèle de processus opérationnel où toute modification d'un objet, c'est-à-dire chaque opération INSERT, UPDATE, ou DELETE, résulte en l'évaluation des déclenchements de transitions entre les états et l'évaluation de conditions spécifiques à chacun d'entre eux, ces deux éléments constituant ainsi le cycle de vie de l'objet.

On peut également se servir de méthodes de *runtime verification*, où la validité de la séquence d'opérations peut être vérifiée en cours d'exécution, au moment même où les opérations sont réalisées par l'intermédiaire d'un organe central. Ceci a notamment été réalisé

par Hallé & Villemaire (2012) dans le contexte de services web. Ici, les auteurs présentent une formalisation de *message contracts*, constitués de l'ensemble des règles qui concernent les séquences valides de messages échangés entre, par exemple, un client et un serveur web. La formalisation originale de ces contraintes en LTL-FO$^+$ (une extension de LTL à la logique du premier ordre) permet l'expression de propriétés *data-aware* (c.-à-d., qui dépendent des données internes des artefacts). À titre d'exemple, dans le cadre de services web de commerce en ligne, Hallé & Villemaire (2012) expriment le fait qu'un client ne puisse supprimer un produit d'un panier qui vient juste d'être vidé (tant qu'un autre produit n'a pas été ajouté) par la formule LTL-FO$^+$:

$$\mathbf{G}\,(\forall_{\mathsf{CartClear/CartID}} c_1 : ((\forall_{\mathsf{CartRemove/CartID}} c_2 : c_1 \neq c_2)\,\mathbf{W}\,(\exists_{\mathsf{CartAdd/CartID}} c_3 : c_1 = c_3)))\,^{9}.$$

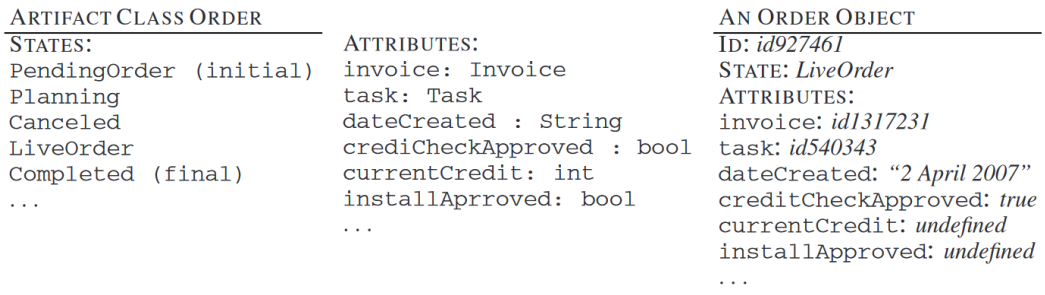
Un moniteur, appelé *watcher*, est ensuite associé à chaque formule LTL-FO$^+$ devant être respectée. Un watcher est un dérivé des automates finis classiques, si ce n'est qu'une *fonction d'évaluation* (*outcome function*) figure en lieu et place des états acceptants. Concrètement, ceci signifie qu'à chaque nouveau message, l'automate déclenche la transition correspondante et évalue si la formule LTL-FO$^+$ associée au watcher est respectée ou non. Il est important de noter qu'avec un tel système, il peut être impossible d'évaluer le respect d'une propriété à un état donné, ce qui ne veut pas dire pour autant que l'exécution ne la satisfait pas (*p. ex.*, la formule $\mathbf{X}\,\varphi$ ne peut être vérifiée qu'à l'état suivant, elle n'est pas violée pour autant à l'état courant). Cette *indécidabilité* implique que la fonction d'évaluation résulte en une des trois valeurs suivantes : $\top$ (c.-à-d., la propriété est satisfaite), $\bot$ (c.-à-d., la propriété est violée) ou ? (c.-à-d., l'évaluation est impossible à cet instant). En outre, les auteurs implémentent ces watchers à l'aide de BeepBeep et les appliquent à plusieurs services web existants.

9. L'opérateur $\varphi\,\mathbf{W}\,\psi$ (« faible jusqu'à ») est similaire à $\varphi\,\mathbf{U}\,\psi$, à ceci près que ψ n'a pas forcement besoin d'être vraie. Dans notre cas, ceci signifie qu'un vidage de panier n'a pas à être nécessairement suivi par l'ajout d'un article dans ce même panier.

La validité des opérations réalisées vis-à-vis de contraintes spécifiques peut même être *assurée* à l'exécution grâce au *runtime enforcement*. Ce dernier reprend les principes de base du runtime monitoring, en observant les événements générés par l'exécution du système considéré. Cependant, un moniteur de runtime enforcement est également capable de *transformer* une séquence (potentiellement invalide) d'événements produite par le système en une autre séquence qui respectera *forcément* la spécification de ce dernier (Schneider, 2000; Falcone, 2010; Falcone *et al.*, 2011). Cette approche est particulièrement étudiée dans le cadre de propriétés *temporisées* (*timed properties*) (Pinisetty *et al.*, 2012; Falcone *et al.*, 2016; Falcone & Pinisetty, 2019), qui spécifient non seulement l'ordre dans lequel les événements issus d'une exécution doivent avoir lieu, mais également le *temps physique* (c.-à-d., quantifié par des unités de temps) qui doit séparer l'apparition de certains types d'événements. Le runtime enforcement dans ce contexte permet alors de *corriger* une séquence invalide en *retardant* des événements s'ils arrivent trop tôt par rapport à la spécification, ou en supprimant les événements dans le cas où aucun retard ne permettrait de satisfaire cette dernière. Le runtime enforcement a également été suggéré, *p. ex.*, pour l'application de contraintes de cycle de vie sur des tags RFID passant d'un lecteur à un autre dans un scénario de *supply chain* (Ouafi & Vaudenay, 2009).

2.2.3 VÉRIFICATION STATIQUE

Dans d'autres cas, connaître le déroulement des tâches à réaliser à l'avance permet d'analyser statiquement le système et de s'assurer que toutes les contraintes déclaratives du cycle de vie sont respectées à chaque étape. C'est le principe de la *vérification statique*, qui assure que l'exécution d'un système respectera toujours les propriétés spécifiées en fournissant une preuve (*p. ex.*, une certification) que le système a bien été vérifié et est conforme à la spécification (la Section 2.3 propose notamment une discussion sur les garanties impliquées



(a) Définition du type d'artefact Order (à gauche) et exemple d'instance (à droite).

SERVICE *UpdateCredit*
WRITE: $\{x$: Order$\}$
READ: $\{y$: CreditReport$\}$
PRE: $\neg$DEFINED(x, creditCheckApproved)
$\wedge\neg$DEFINED(x, currentCredit)
EFFECTS:
– DEFINED(x, creditCheckApproved)
– DEFINED(x, creditCheckApproved)
$\wedge$ DEFINED(x, currentCredit)

(b) Service applicable à un artefact de type Order.

FIGURE 2.13 : Exemple d'artefact de type Order et de service applicable à ce type d'objet, de Bhattacharya *et al.* (2007). Reproduit avec la permission de Springer.

par la vérification statique). Les travaux suivants proposent des méthodes pour appliquer une telle vérification.

Par exemple, Bhattacharya *et al.* (2007) proposent une méthode de modélisation de processus reposant sur l'invocation de *services* sur des *classes* d'artefact. Une classe d'artefact est caractérisée par les états possibles pour celui-ci et des attributs ; une instance d'artefact a donc un identifiant propre ainsi qu'un état défini, et des valeurs sont assignées à ses attributs. Ces derniers sont des éléments d'un ensemble composé à la fois de types primitifs (*p. ex.*, entiers, booléens, chaînes de caractères) et d'identifiants potentiels vers d'autres artefacts du processus. À titre d'illustration, la Figure 2.13a présente un type d'artefact « Order » et en donne un exemple d'instance. Il est possible qu'un attribut ne soit pas défini à des instants

donnés, mais qu'il l'ait été auparavant ou qu'il le soit dans le futur de l'exécution du processus. En outre, les auteurs introduisent la notion de *services*, des composants qui agissent sur les artefacts. Par leur intermédiaire, il est donc possible de créer des artefacts, d'en définir les attributs (ou, à l'inverse, d'en « enlever » la définition) et d'en modifier l'état courant. Ces services sont exécutés si les potentielles pré-conditions spécifiques au service invoqué sont respectées. À titre d'exemple, la Figure 2.13b montre un service *UpdateCredit* applicable à un artefact de classe « Order », avec des pré-conditions et des actions concrètes sur l'artefact, spécifiées respectivement dans les sections « PRE » et « EFFECTS ». Le cycle de vie du modèle consiste alors en des règles explicitant les transitions entre les états des artefacts et les invocations successives de services, selon des conditions sur l'état courant et sur le caractère défini ou indéfini des attributs des artefacts concernés. Les services peuvent être assimilés aux *services web*, services fournis par des serveurs web (*p. ex.*, méthodes POST, PUT), à travers lesquels les modèles de données (*p. ex.*, les entrées d'une base de données) sont manipulés, sous réserve que les requêtes associées respectent des conditions prédéfinies. Calvanese *et al.* (2009) adoptent une approche similaire, où les services ont des pré et post-conditions intra-artefact.

Toujours dans le contexte des services web, Hallé *et al.* (2009) proposent une modélisation sous la forme d'une structure de Kripke dont le cycle de vie consiste en un ensemble de formules CTL-FO$^+$. Le CTL-FO$^+$ est une extension du CTL permettant d'utiliser les quantificateurs classiques de logique de premier ordre au sein des formules CTL, ce qui rend possible l'expression de contraintes sur des valeurs de données ; on parle alors de propriétés *data-aware*. Ce type de considérations résulte généralement en l'explosion exponentielle de la complexité liée à la vérification du modèle ; dans cet article, il est cependant démontré que la vérification de formules CTL-FO$^+$ sur des domaines de valeurs bornés est comparable à celle de formules LTL qui, elles, ne permettent pas l'expression de propriétés *data-aware*.

Les auteurs montrent qu'il est également possible de transposer les formules CTL-FO$^+$ en formules CTL *réduites*, entraînant ainsi une vérification plus rapide dans certains cas.

Gonzalez *et al.* (2012), quant à eux, présentent une méthodologie pour *convertir* des modèles GSM d'artefacts en systèmes de transition, sur lesquels on peut appliquer les méthodes de model checking. Ils implémentent ainsi un outil permettant d'effectuer cette conversion et de vérifier sur le système de transition résultant la validité de propriétés exprimées en CTL. Le processus est complètement automatique et nécessite seulement de fournir en entrée le modèle GSM au format XML ainsi qu'un fichier texte contenant les formules CTL à respecter. Cependant, il est important de noter que l'outil est lui-même soumis à de fortes contraintes puisque les attributs du modèle de données des artefacts sont restreints à seulement quelques types primitifs (énumération, entiers bornés et booléens) et qu'une seule instanciation est possible pour chaque type d'artefact défini dans le système.

2.2.4 APPROCHES DÉCENTRALISÉES

Les méthodes de vérification vues plus tôt ont surtout vocation à être utilisées dans un contexte de système centralisé, en appliquant le model checking ou le runtime monitoring directement sur celui-ci. Pour appliquer un monitoring décentralisé, il serait donc souhaitable que chaque intervenant d'un processus puisse appliquer le runtime monitoring pour vérifier que les contraintes du cycle de vie soient respectées par les actions émises par l'ensemble des intervenants. Les travaux suivants proposent des méthodes qui vont dans ce sens.

RUNTIME MONITORING DÉCENTRALISÉ

Il est notamment possible de réutiliser des notions de runtime monitoring *décentralisé* (Bauer & Falcone, 2012, 2016), conçu pour superviser des systèmes décentralisés. Un tel

système est constitué d'un ensemble de composants $C = \{C_0, \ldots, C_n\}$, où chaque C_i, $i \leq n$, est responsable d'opérations qui lui sont propres. En d'autres termes, ceci signifie qu'un composant C_i est capable d'émettre des événements composés d'un ensemble de propositions atomiques AP_i qui lui est spécifique : $\forall i, j \leq n, i \neq j, AP_i \cap AP_j = \emptyset$. Par exemple, dans une voiture, un capteur de vitesse est responsable de mesurer la vitesse du véhicule et peut émettre un événement *speed_low* (c.-à-d., la vitesse ne dépasse un certain seul prédéfini) ; de façon similaire, des capteurs de pression peuvent émettre des événements *pressure_sensor_k_high* (avec l'identifiant k correspondant au capteur émetteur) lorsqu'ils détectent qu'un utilisateur est assis sur le siège qui leur est associé.

Bauer & Falcone (2012, 2016) considèrent des systèmes décentralisés devant respecter un cycle de vie spécifié par une formule LTL φ. La décentralisation implique que l'ensemble des événements ne soit pas remonté à un noyau central décideur, mais plutôt que les différents composants collaborent pour évaluer le respect de φ. Ainsi, les auteurs associent à chaque C_i un moniteur M_i responsable d'observer la séquence d'événements produite par C_i et d'évaluer si celle-ci satisfait φ. Or, comme nous l'avons mentionné plus tôt, un C_i ne produit des événements ne contenant que certains types de propositions atomiques (soit σ_i un événement produit par C_i, $\sigma_i \in 2^{AP_i}$), son moniteur M_i ne peut donc pas évaluer le respect de φ. En effet, puisque φ est une propriété globale du système, il est fortement probable que celle-ci soit constituée en partie de propositions atomiques n'appartenant pas AP_i. Pour résoudre ce problème, il faut tout d'abord faire *progresser* la formule LTL. Le but de la progression est de diviser la formule en une première formule exprimant ce que l'observation actuelle du composant doit satisfaire, et une deuxième formule, appelée *obligation*, qui doit être satisfaite par la trace d'événements dans le futur.

À un instant $t = 0$, un moniteur local M_i évalue l'événement $\sigma_i(0)$. Si ce dernier ne contient pas certaines des propositions atomiques présentes dans φ, alors M_i garde en mémoire

les parties de la formule correspondantes et fait progresser φ en son obligation φ_i^1. M_i envoie ensuite φ_i^1 aux moniteurs que cette dernière concerne [10], qui peuvent, eux, évaluer au moins en partie φ_i^1 à $t = 2$, la faire progresser, et ainsi de suite jusqu'à ce que chaque moniteur ait pu valider les progressions successives des formules reçues. Cette approche implique nécessairement un délai entre la génération des événements et la décision de satisfaction des propriétés par les moniteurs, mais Bauer & Falcone (2012, 2016) montrent que celui-ci peut être acceptable selon les scénarios, en plus de permettre une réduction des quantités de données communiquées par rapport à un monitoring centralisé, où les composants seraient obligés de transmettre en permanence leurs événements au moniteur central.

Néanmoins, il est important de noter que cette approche, par le biais des progressions successives, peut provoquer une augmentation significative des tailles des formules LTL à vérifier si le comportement spécifié fait intervenir de nombreux composants différents. Cet inconvénient est loin d'être négligeable, en particulier pour les dispositifs limités en mémoire. Pour remédier à ce problème, Falcone *et al.* (2014) ont adapté cette méthode en utilisant des automates en lieu et places des formules LTL. Cette solution permet de réduire la mémoire utilisée comparée à celle utilisant les formules LTL mais, en contrepartie, augmente le nombre de messages échangés entre les composants et, plus faiblement, le délai global de la vérification vis-à-vis de l'exécution.

RUNTIME MONITORING COOPÉRATIF

Dans les processus basés au moins en partie sur la communication de *messages* entre entités (*p. ex.*, diagrammes BPMN, les interactions entre clients et services web), ceux-ci sont souvent considérés (pour des raisons de simplification) comme des données atomiques.

10. Une formule ψ concerne un moniteur M_j si $\text{Prop}(\psi) \subseteq AP_j$, où la fonction $\text{Prop} : \text{LTL} \to 2^{AP}$ renvoie l'ensemble des propositions atomiques constituant une formule LTL.

Or, comme nous avons déjà pu l'observer dans le travail de Hallé & Villemaire (2012), cette approximation est rarement vraie puisque la réception/envoi des messages doit, dans la plupart des cas, respecter un ordonnancement spécifié au préalable par un *contrat*. En outre, dans le contexte particulier des services web, même si ce dernier aspect est bel et bien spécifié, il n'empêche qu'un client et un serveur sont des entités distinctes, exécutant leur propre implémentation de cette spécification. De façon intentionnelle ou non, il est donc tout à fait possible que l'un envoie à l'autre un message inattendu. Le plus souvent, la charge revient donc au serveur de vérifier que le message reçu respecte bien le contrat établi.

En introduisant le runtime monitoring coopératif (ou CRM, pour *Cooperative Runtime Monitoring*), le but de Hallé (2013) est de transférer une partie de cette charge au client. En retour, ce dernier doit fournir une preuve au serveur que le message envoyé est bien valide dans le contexte d'une certaine spécification, ici exprimée à l'aide de formules LTL (la spécification n'est cependant pas restreinte à ce langage). Le fonctionnement du CRM peut être résumé par le processus suivant. D'abord, le client utilise une fonction γ pour produire une preuve que le message m satisfait l'état actuel de l'échange spécifié par φ. Cette preuve et le message sont transmis au serveur. Ce dernier effectue une vérification en deux temps. Tout d'abord, il vérifie que la preuve transmise correspond à m avec la fonction μ. Si $\mu(m, preuve) = \bot$ (c.-à-d., la preuve ne correspond pas à m), alors le message est refusé. Sinon, le serveur confronte la preuve à φ à travers la fonction ν. Si $\nu(preuve, \varphi) = \top$, alors le message respecte bien la spécification et il est accepté.

Tout l'enjeu du système repose bien entendu dans le mécanisme de génération/vérification de la preuve. Pour ce faire, Hallé (2013) s'appuie un algorithme permettant de vérifier *à la volée* si une trace (potentiellement en cours de construction) respecte une formule LTL ou non. Cet algorithme consiste en la décomposition successive de la formule selon les composantes qui doivent être respectées à l'état courant et celles qui doivent l'être à l'état d'après. À

titre d'exemple, considérons une application de l'algorithme pour laquelle l'état initial est $\mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))$ et le message m à envoyer contient les propositions atomiques p, q et $\neg s$. L'opérateur $\mathbf{G}$ signifiant que la formule doit tenir à tous les états, l'état courant doit respecter $p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s)$, et l'état suivant doit satisfaire $\mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))$. Dans le nœud suivant, on sépare donc les deux formules par le symbole $\Vdash$. L'opérateur $\wedge$ signifie que les deux sous formules p et $\mathbf{X}\,q \vee \mathbf{F}\,s$ doivent être satisfaites, sans impact sur l'état suivant. On remplace le $\wedge$ dans le nœud suivant par une virgule pour indiquer que le message doit respecter l'ensemble de ces deux formules. Dans notre message m, p est vrai ; la satisfaction de la formule repose donc sur $\mathbf{X}\,q \vee \mathbf{F}\,s$, qui est isolée, et ainsi de suite. On arrête le développement d'une branche lorsqu'il n'y a plus de formule à la gauche du $\Vdash$. On voit que l'une des branches finit forcément par un $\bot$ à cause de $\neg s$, on sait donc que celle-ci ne pourra jamais être vraie dans les états futurs. Tous les nœuds qui n'ont pas de verdict $\bot$ représentent donc l'ensemble des états à respecter par les messages suivants. Dans notre cas, il s'agit des états $\{q, \mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))\}$ et $\{\mathbf{F}\,s, \mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))\}$. La preuve est formée de ces états et des décompositions en opérateurs ou propositions atomiques menant à eux ; dans cet exemple, on a donc :

$$\begin{aligned} \mathit{preuve} = \gamma(m,\varphi) = \{&\mathbf{G}\,, \wedge, p, \vee_1, \mathbf{X} : \{q, \mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))\}, \\ &\mathbf{G}\,, \wedge, p, \vee_2, \mathbf{F}_2 : \{\mathbf{F}\,s, \mathbf{G}\,(p \wedge (\mathbf{X}\,q \vee \mathbf{F}\,s))\}\}. \end{aligned}$$

La preuve, une fois calculée par le client, et le message sont ensuite envoyés au serveur. Ce dernier peut s'assurer que la preuve correspond bien au message en comparant les propositions atomiques qui y sont présentes ; une différence résultera en un $\mu(m,\mathit{preuve}) = \bot$. La vérification de la preuve se fait ensuite par l'intermédiaire de ν, qui applique les « chemins » suivis par la preuve pour atteindre les états spécifiés. Si les résultats des chemins correspondent, alors la preuve est valide et les états qu'elle contient constituent l'ensemble des états à valider par le message suivant. Ainsi, le serveur a la garantie que le message correspond

bien à l'état courant du contrat sans avoir eu à calculer la totalité des branches d'exécutions, contrairement au client. À travers une implémentation utilisant BeepBeep, l'auteur montre que le CRM permet de réduire le temps de vérification nécessaire au serveur de 29% à 44% selon les propriétés. Il note, néanmoins, que le gain de temps est optimal pour des formules LTL particulières, qualifiées de *non-branching*, pour lesquelles l'application de l'algorithme vu précédemment résulte en une seule et unique branche valide.

2.3 BILAN DES SOLUTIONS EXISTANTES

Dans ce chapitre, nous avons présenté les travaux existants traitant d'une part de la spécification, et d'autre part de la vérification d'un système ou processus par rapport à cette spécification. Étant donné leur variété et leur quantité, il nous semble pertinent de tirer un résumé de ces approches. Puis, nous mettrons en évidence les potentielles limites de ces solutions, motivant ainsi le travail effectué dans cette thèse.

2.3.1 SPÉCIFICATION

Comme nous avons pu le voir dans la section Section 2.1, le sujet de la spécification de cycle de vie de processus est un domaine qui a été substantiellement étudié depuis les débuts de l'informatique. On peut constater cependant que la notion de cycle de vie a beaucoup évolué et, même de nos jours, peut légèrement changer selon le contexte. En outre, le Tableau 2.1 résume les approches mentionnées et détaille pour chacune les propriétés exprimables avec une telle spécification de cycle de vie. Ces propriétés sont réparties en trois catégories :

— l'ordonnancement externe, qui définit les séquences d'événements valides uniquement vis-à-vis des actions externes à l'artefact (c.-à-d., qui ne prennent pas en compte ses données internes) ;

— l'ordonnancement centré sur l'artefact qui, au contraire de l'ordonnancement externe permet de prendre en compte les valeurs des données internes à l'artefact pour constituer les séquences d'exécution valides ;

— le contrôle d'accès, qui contraint la manipulation de l'artefact selon l'entité ou le pair qui l'effectue.

Par exemple, nous avons observé que les automates finis permettaient de décrire uniquement l'ordonnancement de la séquence d'événements produits par un système ou appliqués à un artefact. Les automates représentent d'ailleurs une base sur laquelle de nombreux formalismes tels que les diagrammes de Harel, le BPMN, ou le BEDL s'appuient, au moins en partie.

Cette séquence de tâches donne cependant peu d'indications, voire aucune, sur le contenu des artefacts manipulés par le système, ce qui peut représenter un manque majeur selon les scénarios. Ainsi est né le besoin de spécifications basées sur les artefacts et leurs données internes. Dans ce contexte, un artefact peut être vu comme un *objet* pour lequel sont établies certaines règles concernant sa manipulation et celle de ses données par les entités du processus. C'est notamment le cas des spécifications par BEDL, LTL-FO^{+}, μL, ou GSM. En outre, on peut constater que le BEDL est la seule spécification permettant d'exprimer nativement des politiques de contraintes d'accès, en plus de tout le reste. Ceci est réalisé par l'adoption d'un mécanisme semblable au *Role Based Access Control* (RBAC) (Ferraiolo & Kuhn, 1992), qui donne aux participants du processus des rôles auxquels sont associés des droits (*p. ex.* le rôle *A* peut lire la donnée *x* de l'artefact, lire/modifier *y*, et créer *z*).

Spécification	Propriétés exprimables			Références
	Ord. ext.	Ord. centré sur l'art.	Accès	
Automates finis	X			Hopcroft *et al.* (2006); Rosen (2011); Gopalakrishnan (2019)
Réseaux de Petri	X			Petri (1962); Petri & Reisig (2008)
LTL	X			Pnueli (1977, 1979)
LTL-FO$^+$(extension de la LTL)	X	X		Hallé & Villemaire (2012); Khoury *et al.* (2016)
Diagrammes de Harel	X			Harel (1987)
UML (diagrammes d'activité et d'états-transition)	X			Object Management Group (2017)
BPMN	X			Object Management Group (2011)
BEDL	X	X	X	Nandi *et al.* (2010)
μL (extension du μ-calcul)	X	X		Hariri *et al.* (2011)
GSM	X	X		Hull *et al.* (2011)

TABLEAU 2.1 : Résumé et comparaison des approches de spécification de cycles de vie vues en Section 2.1 © Quentin Betti.

2.3.2 APPLICATION

Dans la Section 2.2, nous avons pu voir que faire respecter un cycle de vie, ou l'appliquer, consistait à vérifier que le système considéré était bien conforme vis-à-vis de celui-ci. Cette vérification peut se faire principalement de deux manières différentes : le model checking ou le runtime verification. Le model checking consiste en l'abstraction du système par un modèle spécifiant l'ensemble des exécutions possibles. La vérification revient alors à vérifier que le modèle respecte bien le cycle de vie spécifié. À l'inverse, dans le cas du runtime verification, il s'agit de monitorer (c.-à-d., observer) l'exécution du système en temps réel en se basant sur la trace d'événements qu'il produit. Dans ce cas, pour appliquer le cycle de vie, il suffit de s'assurer que la trace d'événements respecte bien sa spécification. Nous avons également présenté BeepBeep, une bibliothèque de CEP, permettant notamment l'implémentation de moniteurs de runtime verification.

Par la suite, nous avons listé de nombreux travaux existants utilisant ces méthodes pour appliquer un cycle de vie ; ceux-ci sont résumés dans le Tableau 2.2, en indiquant, pour chacun, la méthode de vérification et la spécification de cycle de vie utilisées. Nous avons réparti ces approches en trois catégories :

— les approches centralisées, où toutes les interactions sur les artefacts sont réalisées à travers un organe central, aussi appelé *worflow engine*. Dans ce contexte, vérifier la conformité d'un processus par rapport à un cycle de vie consiste à appliquer le model checking ou le runtime verification directement sur le système central (parfois les deux) ;

— les approches par vérification statique, pour lesquelles le but est de certifier que les systèmes des intervenants du processus sont conformes au cycle de vie. Ceci est principalement fait en appliquant le model checking aux systèmes considérés ;

Catégorie de l'approche	Type de l'approche	Travaux	Spécification du cycle de vie
Centralisée	MC	Zhao *et al.* (2009)	TiLE
		Gerede & Su (2007)	ABSL
		Ataullah & Tompa (2011)	Automates et conditions dans une BDD
	RV	Hallé & Villemaire (2012)	LTL-FO$^+$
	RE	Pinisetty *et al.* (2012)	Automates temporisés
		Falcone *et al.* (2016)	
		Falcone & Pinisetty (2019)	
		Ouafi & Vaudenay (2009)	Automates finis
Vérification statique	MC	Bhattacharya *et al.* (2007)	Logique de premier ordre
		Calvanese *et al.* (2009)	
		Hallé *et al.* (2009)	CTL-FO$^+$
		Gonzalez *et al.* (2012)	CTL
Décentralisée	RV	Bauer & Falcone (2012)	LTL
		Bauer & Falcone (2016)	
		Hallé (2013)	LTL-FO$^+$

TABLEAU 2.2 : Résumé des travaux sur l'application de cycles de vie vus en Section 2.2 (MC = model checking, RV = runtime verification, RE = runtime enforcement) © Quentin Betti.

— les approches décentralisées, qui permettent aux différentes entités du processus de collaborer dans le runtime verification de son exécution.

2.3.3 MOTIVATIONS DU PRÉSENT MANUSCRIT

Bien que les travaux concernant la spécification de cycle de vie et la vérification de la conformité de processus soient nombreux, on peut tout de même souligner certains éléments peu ou pas du tout traités par les solutions présentées plus tôt.

PROPRIÉTÉS EXPRIMABLES

Premièrement, on peut constater que les propriétés exprimables par les différentes spécifications sont en partie limitées. Par exemple, on peut noter que seul le BEDL permet nativement l'expression de politiques de contrôle d'accès, en plus des différents ordonnancements des séquences d'événements. De plus, bien que les spécifications propices à l'expression de propriétés data-aware soient relativement bien étudiées, aucune ne fait référence au format même des données de l'artefact. Il serait appréciable, en effet, de pouvoir également préciser qu'une certaine manipulation d'artefact soit respectueuse du format de ses données. Ce format serait défini *a priori* de l'exécution du processus, tout comme les autres contraintes, l'incluant à part entière dans le cycle de vie de l'artefact. En d'autres termes, au cours d'un processus, un pair ne devrait pas être autorisé à, par exemple, modifier un champ correspondant à un numéro de téléphone en y insérant une valeur ne satisfaisant pas le format attendu d'un numéro de téléphone. Par la suite, nous appellerons ces propriétés des *contraintes d'intégrité*.

FLEXIBILITÉ

Nous avons également pu voir que, parmi les méthodes de vérification exploitées, nombre d'entre elles dépendaient d'un langage ou formalisme particulier de spécification de cycle de vie. Il s'agit là d'un manque majeur de flexibilité puisque les méthodes ne peuvent pas s'adapter aux propriétés exprimées dans d'autres langages. Dans le cas des approches utilisant le model checking, le problème est même exacerbé puisque, pour que la vérification d'un modèle soit facilement réalisable (voire simplement réalisable), de sévères restrictions doivent être imposées sur les propriétés pouvant être exprimées, ou sur la complexité sous-jacente de l'environnement d'exécution. En outre, rien que le fait de déterminer si le problème de vérification est résoluble ou non est devenu un sujet de recherche en lui-même. Par exemple,

Zhao *et al.* (2009) et Calvanese *et al.* (2009) considèrent des modèles d'artefacts dont les cycles de vie sont exprimés en logique du premier ordre uniquement et où les exécutions sont finies. Les approches de Bhattacharya *et al.* (2007) et Gerede & Su (2007) imposent que les domaines de définition des données soient bornés, ou que les pré et post-conditions se réfèrent seulement aux artefacts, et non aux valeurs de leurs variables (Gerede & Su, 2007).

En réalité, seule l'approche du CRM de Hallé (2013) évoque la possibilité d'être compatible avec tout type de spécification, pourvu que leur vérification soit NP-complète. Néanmoins, ce cas présente tout de même une flexibilité limitée puisque la modification du langage utilisé impliquerait nécessairement des changements dans le processus de calcul de preuve, l'algorithme mentionné étant prévu uniquement pour des formules LTL et LTL-FO$^+$. Là aussi, il serait intéressant de concevoir une méthode faisant complète abstraction de la manière dont est spécifié le cycle de vie.

BESOIN DE CONFIANCE ET DÉCENTRALISATION

Enfin, parmi les solutions de vérification existante, nombreuses sont celles nécessitant une part de *confiance*. Pour les approches centralisées, celle-ci est évidente : les pairs du processus doivent avoir une confiance totale envers l'organe central à travers lequel ils effectuent les manipulations d'artefacts. Si celui-ci est compromis, il est alors impossible de savoir si la vérification effectuée est celle escomptée, ou même si elle a simplement lieu. Cela est peut-être moins évident pour les approches par vérification statique. En effet, il serait tentant de penser que, à partir du moment où les systèmes des pairs collaborant sont vérifiés et certifiés, alors toute action émanant de l'un d'eux est forcément valide et que, par extension, aucun contrôle supplémentaire n'est nécessaire. Néanmoins, rien ne permet de totalement garantir qu'un pair n'ait pas réussi à modifier son système sans remettre en cause sa certification, ou qu'il n'ait

pas pu la contourner, tout simplement. Un tel scénario compromettrait alors l'ensemble du processus et n'est en aucun cas souhaitable.

D'un autre côté, nous avons pu voir que la décentralisation effective de la vérification, où aucun organe central n'est requis, propose des solutions limitées. Le CRM, par exemple, ne protège pas les messages et preuves échangés entre un client et un serveur ; une requête peut être remplacée par une autre, grâce à une attaque de l'homme du milieu (*man-in-the-middle attack*), et acceptée par le serveur tant que c'est une continuation valide de l'échange de messages actuel. De plus, cette approche est restreinte à une unique communication point à point et unidirectionnelle ; c'est-à dire que la communication est effectuée dans le cadre d'une paire unique de client-serveur, et seul ce dernier ne requiert pas de confiance préalable envers le client. L'approche du monitoring décentralisé de Bauer & Falcone (2012, 2016) permet d'échanger des messages entre plusieurs entités de façon multidirectionnelle mais, dans ce cas, toutes les actions envoyées sont jugées légitimes par l'ensemble des composants, sans jamais être remises en cause. Le moniteur compromis d'un capteur pourrait, par exemple, envoyer des propositions atomiques caractérisant des fonctionnalités ou contextes qui ne le concernent pas, et ce sans que les autres moniteurs s'en aperçoivent, provoquant potentiellement des évaluations de propriétés illégitimes. Ces dernières pourraient avoir des conséquences graves selon les scénarios ; dans le cas des composants d'une voiture, ceci pourrait, par exemple, déclencher l'activation soudaine du système de freinage ou de certains signaux lumineux.

PROPOSITIONS DU PRÉSENT MANUSCRIT

Dans cette thèse, il s'qgit de mettre en place une méthode permettant une application de cycle de vie respectant les critères suivant.

— La méthode doit être flexible vis-à-vis de la spécification de cycle de vie et de la définition de l'artefact concerné. En l'occurrence, nous verrons dans le Chapitre 5 que les artefacts observés sont des documents sur lesquels nous exprimons des propriétés sur l'ordre (qu'il fasse référence aux données internes à l'artefact ou à des événements extérieurs), l'intégrité des données du document (c.-à-d., leur format, comme nous l'avons défini plus tôt), et les politiques de contrôle d'accès. Nous ne restreignons cependant ni l'expression du cycle de vie à ces types de propriétés, ni la définition de l'artefact à ce type de document, et envisageons ces derniers comme des « boîtes noires » ;

— Les manipulations sur les artefacts doivent être réalisées de façon décentralisée, sans passer par un serveur ou toute autre forme de centralisation. Ainsi, les pairs intervenant dans le processus s'envoient eux-mêmes les manipulations effectuées ;

— La décentralisation des manipulations ne doit pas remettre en cause leur sécurité. Cela signifie concrètement que :

  — les auteurs des manipulations doivent être identifiés de manière à ce qu'aucune usurpation d'identité ne soit possible ;

  — toute manipulation effectuée ne doit pas pouvoir être remise en cause (c.-à-d., modifiée ou supprimée) ;

  — le contenu des manipulations doit être confidentiel, pour qu'une entité extérieure au processus ou qu'un pair non autorisé ne puisse pas accéder au contenu d'une partie ou de la totalité des manipulations réalisées sur l'artefact.

Pour ce faire, nous considérons, dans un premier temps, l'adoption d'une approche basée sur la *blockchain*, une technologie encore émergente et considérée comme particulièrement prometteuse dans les contextes faisant intervenir des processus collaboratifs. Ainsi, nous

présentons le fonctionnement de la blockchain dans le prochain chapitre et un cas d'utilisation dans le Chapitre 4. Nous verrons cependant qu'une telle technologie présente certains inconvénients, et nous introduirons dans le Chapitre 5 une approche originale qui nous semble plus pertinente dans ce type d'application.

CHAPITRE III
LES TECHNOLOGIES BLOCKCHAIN : ÉTUDE VIA LA PLATEFORME ETHEREUM

La blockchain est un cas particulier des approches décentralisées pouvant être employée dans l'application de cycle de vie. Elle permet aux pairs de transporter et stocker des transactions de manière distribuée, sans aucun tiers superviseur. Le but de ce chapitre est de décrire les fondements techniques des technologies de type blockchain afin d'en identifier les fonctionnalités clés et leurs limites. Il existe de nombreuses plateformes et implémentations de réseau blockchain : dans le seul domaine de la cryptomonnaie, on recense plus de 2500 devises distinctes (CoinMarketCap, 2019) reposant sur des implémentations plus ou moins variées. Bien que les concepts de base soient les mêmes, il n'empêche que certaines plateformes présentent des caractéristiques et des fonctionnalités différentes.

Afin de décrire en détail certains mécanismes, nous aborderons donc tout au long de ce chapitre les particularités de la technologie blockchain utilisée par Ethereum [11], une plateforme open source très populaire puisqu'elle comptabilise près de 100 millions d'adresses uniques actuellement (Etherscan, 2019). Ce système a pour devise l'Ether, la deuxième cryptomonnaie en termes de capitalisation boursière (CoinMarketCap, 2019).

3.1 PRÉLIMINAIRES TECHNIQUES

Avant de présenter en détail les mécanismes et spécificités des blockchains, il est tout d'abord essentiel de définir et expliquer les différentes notions techniques qui leur sont fondamentales. Celles-ci concernent particulièrement le domaine de la cryptographie et cette

11. https ://www.ethereum.org/

section fournit les éléments de base sur le hachage, le chiffrement et les signatures numériques, nécessaires à la bonne compréhension du chapitre.

3.1.1 FONCTIONS DE HACHAGE

Une fonction de hachage est une fonction permettant, à partir d'une donnée binaire de taille arbitraire, de produire une chaîne binaire de taille fixe ; cette dernière est alors appelée valeur de hachage, ou condensat (Menezes *et al.*, 1996). En ce sens, un condensat peut être défini comme le *résumé* compact d'une donnée. Largement utilisé dans le partage de fichiers, le condensat permet alors d'assurer l'intégrité d'une donnée puisqu'il suffit de comparer la valeur de hachage communiquée par l'émetteur et celle calculée directement à partir des données reçues. Si les deux diffèrent, alors les données ont été modifiées, et sont donc potentiellement compromises.

COLLISIONS, TABLES DE HACHAGE ET TECHNIQUES DE RÉSOLUTION

Le scénario où deux entrées distinctes partagent un condensat identique généré par la même fonction de hachage h, c'est-à-dire $h(x) = h(y)$ avec $x \neq y$, est appelé une *collision*. Étant donné le principe des fonctions de hachage selon lequel une donnée de taille arbitraire est transformée en une donnée de taille fixe, les collisions leur sont inévitables. En effet, une fonction produisant des condensats de n bits ne propose donc que 2^n valeurs de hachage différentes, alors qu'il y a potentiellement une infinité [12] d'entrées possibles : même avec une fonction présentant une *distribution* parfaite de valeurs de hachage, il existe donc forcé-

12. Les fonctions de hachage ont en pratique un domaine de définition fini, mais dont la taille est *bien plus* grande que celle de son image. Par exemple, la fonction SHA-1 traite des messages dont la taille peut atteindre jusqu'à $2^{64} - 1$ bits, alors qu'elle produit des condensats de 160 bits (Dang, 2015).

ment des entrées distinctes produisant les mêmes condensats. Dans certains scénarios, il est indispensable d'utiliser une fonction de hachage présentant un très faible risque de collisions.

Prenons, par exemple, le cas des tables de hachage. Une table de hachage est une implémentation spécifique de tableau associatif dans laquelle les condensats des clés sont utilisés comme index (Cormen *et al.*, 2009). Soit h une fonction de hachage, et k la clé associée à une valeur v. Pour ajouter v à une table de hachage, on va en fait créer un tableau T et insérer v à l'index $h(k)$: on a donc $T[h(k)] = v$. Pour récupérer une valeur, il n'y a donc qu'à calculer le condensat de sa clé et utiliser celui-ci comme index pour parcourir le tableau T. Un problème est alors posé lorsqu'une collision a lieu. Si, pour une valeur $v' \neq v$ associée à la clé k', il résulte que $h(k) = h(k')$, alors v et v' partageront le même index dans T. L'ajout de l'une provoquera alors l'écrasement de celle précédemment stockée, ce qui n'est probablement pas souhaitable, d'où l'importance de réduire le plus possible le risque de collision de la fonction de hachage h utilisée.

Néanmoins, comme nous l'avons dit plus tôt, les collisions sont inévitables. Heureusement, il existe plusieurs techniques de résolution pour pouvoir stocker correctement des valeurs dont les index entrent en collision. L'une d'entre elles, appelée résolution par *chaînage* (Cormen *et al.*, 2009), consiste à stocker des *listes chaînées* dans les cases du tableau au lieu des valeurs. Ainsi, deux valeurs partageant le même index sont en fait ajoutées comme éléments de la liste chaînée contenue à cet index ; un élément de la liste contient alors la valeur *et* sa clé (et non le condensat de la clé), permettant donc de distinguer les valeurs par leurs clés lorsque celles-ci produisent des condensats identiques. Ajouter, modifier, lire ou supprimer une valeur dans la table de hachage revient alors à appliquer les opérations classiques d'ajout, de recherche ou de suppression propres aux listes chaînées.

Pour être qualifiée de *cryptographique*, une fonction de hachage h doit cependant respecter deux principales caractéristiques (Menezes *et al.*, 1996). Premièrement, la fonction doit être mathématiquement irréversible : pour un condensat donné y, il doit être impossible de calculer l'entrée x telle que $h(x) = y$. Cela permet notamment d'assurer la confidentialité de l'information dont on aurait stocké le condensat. Dans les distributions Linux, telles que Debian (2019), les mots de passe des utilisateurs ne sont pas stockés directement ; seuls leurs condensats sont conservés. Lorsqu'un utilisateur se connecte, il fournit son mot de passe et le système calcule alors sa valeur de hachage qui est comparée à celle enregistrée lors de la création du compte. Deuxièmement, il doit être mathématiquement impossible de calculer à l'avance deux entrées ayant le même condensat, c.-à-d. calculer x et y distincts tels que $h(x) = h(y)$ (en d'autres termes, il doit être impossible de pré-calculer les entrées donnant lieu à des collisions).

Par exemple, MD5 est un algorithme de hachage cryptographique crée par Rivest (1992). Largement utilisé, il est néanmoins sujet à des failles importantes. Notamment, il est particulièrement peu résistant aux attaques de collision : il est possible en quelques secondes de calculer deux messages entrant en collision (Wang & Yu, 2005), ce qui contredit complètement une des caractéristiques des fonctions de hachage cryptographiques.

À contrario, les SHA (*Secure Hash Algorithms*) constituent une famille d'algorithmes sécurisés, publiés et standardisés par le *National Institute of Standards and Technology* (NIST). L'ensemble de ces algorithmes est réparti, à l'heure actuelle, en deux rapports. Premièrement, Dang (2015) détaille les sous-familles SHA-1 (anciennement SHA-0), et SHA-2 ; cette dernière englobe les algorithmes SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 et SHA-512/256. Pritzker & May (2015), quant à eux, standardisent une « nouvelle » famille

d'algorithme appelée SHA-3, anciennement connue sous le nom de Keccak. Bien qu'elles restent plus sécurisées que le MD5, le NIST (2017) recommande tout de même d'éviter, par exemple, l'usage du SHA-1 pour toute application nécessitant une résistance aux collisions, comme les signatures numériques (définies en Section 5), ou encore d'envisager l'adoption du SHA-256 au minimum pour les applications utilisant la famille SHA-2. La famille SHA-3 est quant à elle utilisable pour tout type d'application.

3.1.2 MÉCANISMES DE CHIFFREMENT

Comme nous l'avons mentionné, le condensat d'une donnée permet d'assurer que celle-ci n'a pas été modifiée, sous réserve que l'on ait utilisé une fonction de hachage sécurisée et que le condensat transmis n'a pas été lui-même falsifié. On dit alors que le condensat assure l'*intégrité* de la donnée. De plus, posséder un condensat ne nous permet pas de retrouver la donnée originelle ou d'en extraire quelconque information (à condition, là encore, que la fonction de hachage utilisée soit suffisamment sécurisée). En outre, préserver la *confidentialité* d'une donnée est généralement souhaitable (*p. ex.*, il ne devrait pas être possible de retrouver un mot de passe à partir de son condensat).

Les mécanismes de chiffrement, eux, permettent d'assurer la confidentialité d'une donnée tout en la partageant aux utilisateurs autorisés. Par exemple, il est possible de chiffrer un appel audio/vidéo, un transfert de fichier ou une messagerie instantanée, tout en gardant les informations échangées *secrètes* vis-à-vis des protagonistes extérieurs à la communication, empêchant ainsi en partie les *attaques de l'homme du milieu*. Nous allons voir, cependant, que les mécanismes de chiffrement permettent également d'assurer l'intégrité et l'authenticité des données, notamment via l'utilisation de *signatures numériques*.

Au cœur des mécanismes de chiffrement repose le partage de *clés*. Ces dernières sont utilisées par l'émetteur pour *déguiser* l'information originale en un bloc de données incohérent appelé *cryptogramme* (*ciphertext* en anglais). Ce sont également les clés qui permettront au destinataire de retransformer le cryptogramme en une donnée cohérente et compréhensible. Le passage de la donnée en clair à son cryptogramme, et inversement, sont respectivement appelés *chiffrement* et *déchiffrement*. Évidemment, ces deux opérations sont liées par l'utilisation d'algorithmes. On observe cependant deux grandes catégories de chiffrement, qui diffèrent par leur utilisation des clés : les mécanismes de chiffrement *symétriques* et *asymétriques*.

CHIFFREMENT SYMÉTRIQUE

On dit qu'un algorithme de chiffrement est symétrique lorsque la clé utilisée pour chiffrer est la même que celle nécessaire pour déchiffrer, ou que l'une est facilement calculable à partir de l'autre (Menezes *et al.*, 1996). Un des plus vieux exemples d'un tel mécanisme est le chiffre de César, aussi appelé chiffrement par décalage. Le principe est simple : au lieu d'écrire un texte tel quel, chaque lettre sera remplacée par celle correspondant à un certain nombre de décalages dans l'alphabet. Ainsi, avec le chiffre de César présenté dans la Figure 3.1, qui utilise un décalage de 2 vers la droite, la lettre « A » devient « C », « B » devient « D », ainsi de suite jusqu'à « Z » qui devient « B » (on revient au début de l'alphabet lorsqu'on atteint la fin). Par exemple, avec un décalage de 13 vers la droite, la phrase « VENI, VIDI, VICI » devient « IRAV, IVQV, IVPV ». Pour déchiffrer le message, il suffit de réaliser le décalage inverse, à savoir un décalage de 13 vers la gauche. Nous avons donc bien un chiffrement symétrique, avec pour clé le décalage de 13 positions. Un tel décalage, appelé ROT13 (Schneier, 1996), est d'ailleurs un cas particulier du chiffre de César, car l'appliquer une nouvelle fois au cryptogramme permet de retrouver le document original (en cause : la symétrie impliquée par un décalage de 13 positions dans un alphabet de 26 lettres).

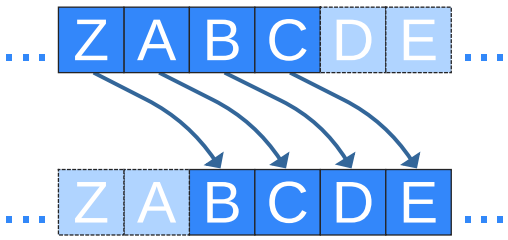


FIGURE 3.1 : Exemple de chiffre de César avec un décalage de 2 vers la droite © Quentin Betti.

Cependant, en utilisant ce mécanisme, il est tout à fait possible de retrouver le message sans avoir la clé : il suffit en effet de tenter les 26 décalages possibles les uns après les autres jusqu'à tomber sur un message cohérent. Il faut donc, au pire, 26 tentatives pour trouver le message ce qui peut représenter quelques minutes à la main, et seulement quelques microsecondes pour un ordinateur récent. Ce faisant, on a donc déchiffré le message et déterminé sa clé par *force brute*, c.-à-d. en essayant toutes les combinaisons possibles jusqu'à trouver la bonne.

Des algorithmes *beaucoup* plus récents, comme AES (pour *Advanced Encryption Standard*) (NIST, 2001), ont été conçus pour être plus résistants aux attaques par force brute tout en restant performant. Détailler le fonctionnement complet d'AES serait un peu hors du propos de ce manuscrit. Néanmoins, on peut décrire AES comme un algorithme de chiffrement *par bloc* fonctionnant sur le principe de *réseau de substitution-permutation* (Shannon, 1949). Concrètement, cela consiste à répartir la donnée à chiffrer en blocs et d'appliquer à ceux-ci des transformations mathématiques successives prenant en compte la clé secrète. Ces opérations sont réalisées en plusieurs *tours* afin de produire un bloc final constituant le cryptogramme. Inverser le processus des opérations avec la bonne clé permet de déchiffrer ce dernier et de retrouver ainsi le message original. On a donc là encore une seule clé permettant à la fois le chiffrement et le déchiffrement, connue seulement des parties impliquées dans la communication.

CHIFFREMENT ASYMÉTRIQUE

Le chiffrement asymétrique se distingue du chiffrement symétrique par l'utilisation d'une clé de chiffrement et d'une clé de déchiffrement (Menezes *et al.*, 1996). La première est appelée clé *publique*, et la seconde clé *privée*. Comme leur nom le sous-entend, la clé publique peut être communiquée à tout le monde, même aux acteurs extérieurs à la communication concernée ; la clé privée quant à elle, doit être gardée secrète et n'être connue que de son possesseur. Le fait que le déchiffrement puisse être effectué avec une clé différente que celle utilisée pour chiffrer la communication est assuré par les algorithmes de chiffrement asymétriques et leurs mécanismes de génération de clé.

De manière générale, les mécanismes asymétriques fonctionnent selon le processus suivant. Alice veut envoyer un message confidentiel à Bob. Ce dernier doit d'abord rendre sa clé publique disponible pour Alice, en lui envoyant directement ou en la publiant dans un annuaire auquel elle peut accéder. Une fois son message écrit, Alice récupère la clé publique de Bob et l'utilise pour chiffrer le message. Elle envoie ensuite le message chiffré à Bob. Jusqu'ici, quiconque intercepterait le message ne mettrait la main que sur des données incompréhensibles. Lorsque Bob reçoit le message, il n'a qu'à utiliser sa propre clé privée pour déchiffrer le message et en lire le contenu. Tenter le déchiffrement avec une autre clé donnerait lieu à un résultat complètement incohérent.

Un des algorithmes de chiffrement asymétrique les plus connus est RSA (Rivest *et al.*, 1978), nommé ainsi après ses trois auteurs, Ronald Lin Rivest, Adi Shamir et Leonard Adleman. Celui-ci se base sur l'algorithme de génération de clés suivant :

1 Choisir deux grands nombres premiers p et q distincts

2 Calculer $n = pq$ et $\phi = (p-1)(q-1)$

3 Choisir un entier aléatoire e, $1 < e < \phi$, tel que $\gcd(e, \phi) = 1$

4 Calculer d, $1 < d < \phi$, tel que $ed \equiv 1 \pmod{\phi}$

5 (e,n) est la clé publique ; d est la clé privée

Algorithme 3.1 : Algorithme de génération de clés RSA.

Il faut tout d'abord choisir deux nombres premiers p et q *suffisamment* grands [13]. Il faut ensuite calculer le *module de chiffrement* $n = pq$, ainsi que $\phi(n) = (p-1)(q-1)$. Un entier e strictement inférieur et premier avec ϕ est alors choisi : il s'agit de l'*exposant de chiffrement*. Finalement, on calcule l'*exposant de déchiffrement* d, strictement inférieur à ϕ, qui est l'inverse de $e \pmod{\phi}$. Les clés privées et publiques sont alors respectivement (e,n) et d (on peut aussi garder le module n dans la clé privée, celle-ci devenant alors (d,n)).

Pour chiffrer un message, il faut d'abord le convertir en une représentation numérique m (*p. ex.*, en remplaçant ses caractères par leurs valeurs en ASCII). Si (e,n) est la clé publique de Bob, Alice doit alors calculer le cryptogramme c :

$$c = m^e \pmod{n}$$

Alice envoie ainsi c à Bob. À sa réception, Bob va utiliser sa clé privée pour déchiffrer le cryptogramme et ainsi récupérer le contenu du message original m :

13. La sécurité du RSA venant principalement de la complexité à factoriser les *très* grands nombres, il faut donc choisir p et q suffisamment grands pour que n, qui fait partie de la clé publique, ne puisse être facilement factorisable (cela permettrait en effet de retrouver la clé privée d).

$$m = c^d \pmod{n}$$

On voit donc que le chiffrement via RSA permet d'assurer la confidentialité d'une information partagée entre deux protagonistes. Cependant cela ne les protège pas contre d'éventuelles modifications malicieuses et usurpations d'identité. Si Mallory, une attaquante ayant accès aux échanges entre Alice et Bob, venait à envoyer un message à Bob en prétendant être Alice, celui-ci n'aurait aucun moyen de se rendre compte de la supercherie. De plus, rien n'empêcherait Mallory d'intercepter le message chiffré d'Alice, d'en modifier le contenu, et de le transmettre à Bob. Ce dernier tenterait de le déchiffrer, retrouverait alors très probablement un contenu illisible et, ne pouvant être au courant de la modification du message, se questionnerait sans doute sur l'incohérence des propos d'Alice. Le chiffrement seul [14] ne permet donc d'assurer ni l'intégrité, ni l'authenticité du message, introduisant alors le besoin de *signatures numériques*.

SIGNATURES NUMÉRIQUES

À l'instar d'une signature manuscrite sur document physique, les signatures numériques ont pour rôle de prouver l'intégrité et l'authenticité d'une donnée auprès de leur destinataire. Concrètement, il s'agit d'une séquence de bits accompagnant la donnée envoyée qui a été générée par l'émetteur pour que son destinataire, en la vérifiant, ait la certitude que la donnée a) ne peut pas avoir été envoyée par une autre personne que l'émetteur, et b) qu'elle n'a pas été modifiée en chemin. Elle permet donc également d'assurer la *non-répudiation* des

14. Certains *modes* de chiffrement, par exemple le GCM (pour *Galois/Counter Mode*) (Dworkin, 2007) permettent tout de même d'assurer la confidentialité et l'intégrité des données dans le cadre de chiffrement par blocs.

données : l'émetteur d'une donnée signée ne peut réfuter le fait qu'il en a été l'auteur, ce qui est particulièrement utile dans un contexte légal.

De manière générale, les signatures sont étroitement liées au chiffrement asymétrique, et leur fonctionnement est réparti sur deux phases : la *génération* et la *vérification*. Supposons qu'Alice veuillent envoyer un message à Bob. Alice va d'abord *générer* la signature associée à son message en utilisant sa clé privée. Lors de sa réception, Bob va pouvoir *vérifier* que la signature a bien été générée par la clé privée d'Alice en utilisant la clé publique de cette dernière.

Par exemple, RSA permet, en plus du chiffrement, de générer et vérifier une signature (Rivest *et al.*, 1978). Il suffit pour cela « d'inverser » les clés dans le processus du chiffrement. Si (e,n) et d sont respectivement les clés publiques et privées RSA d'Alice, alors celle-ci va générer une signature s pour le message m en calculant $s = \widetilde{m}^d \mod n$, où $\widetilde{m}$ est un *équivalent* de m. Concrètement, on choisira souvent $\widetilde{m} = h(m)$, avec h une fonction de hachage cryptographique. Lorsque Bob reçoit le message signé, c.-à-d. le couple (m,s), il doit alors calculer $s' = \widetilde{m}^e \mod n$. Si $s = s'$ alors l'intégrité et l'authenticité du message sont assurées. Sinon, m peut avoir été modifié entre temps, ou une tierce personne a tenté de se faire passer pour Alice. On peut évidemment combiner le chiffrement et la signature d'un message ; il suffirait alors de remplacer le message par son cryptogramme dans les étapes citées plus tôt.

D'autres algorithmes peuvent être utilisés pour générer des signatures. Par exemple, ECDSA, pour *Elliptic Curve Digital Signature Algorithm* (Johnson *et al.*, 2001), est un algorithme faisant partie de la famille de cryptographie sur *courbes elliptiques*, abrégée ECC (pour *Elliptic Curve Cryptography*) et introduite par Miller (1986) ainsi que Koblitz (1987). Dans le cadre de l'ECC, les courbes elliptiques sont des courbes planes correspondant à l'équation $y^2 = x^3 + ax + b$ définies sur un *corps fini*. L'ECC repose sur l'arithmétique

géométrique, plus particulièrement sur l'addition et la multiplication de points géométriques de courbes elliptiques.

3.2 UNE STRUCTURE EN BLOCS

Les principaux aspects techniques nécessaires ayant été présentés à la section précédente, nous pouvons maintenant revenir au fonctionnement des blockchains. Ces dernières ont été introduites pour la première fois en 2008, avec une célèbre preuve de concept : le Bitcoin (Nakamoto, 2008), première d'une longue série de cryptomonnaies. À travers cette publication originale, Nakamoto [15] détaille ce qui sera la base des blockchains, à savoir leur structure en blocs.

Trivialement, une blockchain est une structure de données composée de *blocs* de *transactions*. Comme on peut le voir dans la Figure 3.2, les blocs sont chaînés : chacun d'entre eux référence le condensat de son prédécesseur dans son en-tête, à l'exception du tout premier, souvent appelé *bloc 0* ou *bloc genesis*.

3.2.1 BLOC GENESIS

Le bloc genesis est le premier bloc de la blockchain. Dans le cas d'Ethereum, à chaque fois qu'un nouveau client souhaite se connecter au réseau, il recrée ce bloc pour se synchroniser avec le reste du réseau. Les paramètres de la création du bloc genesis sont généralement spécifiés dans un fichier de configuration `genesis.json`, excepté dans le cas d'une connexion au Mainnet [16] d'Ethereum (le réseau principal, sur lequel la majorité des transactions est

15. Nakamoto n'est en réalité qu'un pseudonyme, la réelle identité de l'auteur du livre blanc du Bitcoin restant un mystère, et ce malgré quelques suppositions (Alvarez, 2019; Sharma, 2019).

16. https://etherscan.io/

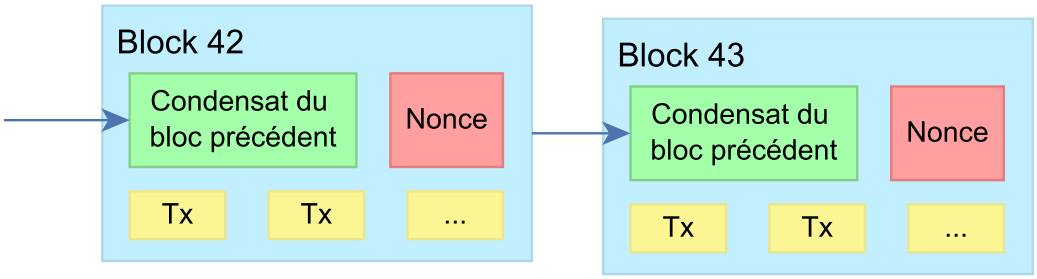


FIGURE 3.2 : Structure générale des blockchains, adaptée de Betti *et al.* (2020). Reproduit avec la permission de Springer.

effectuée) ou de quelques réseaux de test populaires comme Kovan [17] ou Rinkeby [18], dont les configurations sont directement inclues dans le code source des clients Ethereum.

Pour que des clients se synchronisent sur le même réseau, il faut impérativement que les blocs 0 générés, et donc les paramètres de leur création, soient identiques. Entre autres, le bloc genesis va spécifier certaines caractéristiques du réseau, notamment l'allocation initiale d'Ether à certains comptes ou la difficulté de base du *minage* de blocs (voir Section 3.4.3).

3.2.2 BLOCS "STANDARDS"

Une fois le bloc genesis construit, les blocs sont ajoutés les uns à la suite des autres, comme pour une liste chaînée standard. Cependant, à l'inverse de ces dernières, lorsqu'un bloc est inséré, il ne peut être ni modifié ni supprimé, faisant de la blockchain un registre immuable de transactions. Cette immuabilité est rendue possible par l'utilisation de fonctions de hachage dans la référence au bloc précédent.

17. https://kovan.etherscan.io/
18. https://rinkeby.etherscan.io/

Dans le cas d'Ethereum, un bloc est composé, entre autres, d'un en-tête et d'un corps contenant les transactions (Wood, 2019). Concrètement, un en-tête est une collection de champs contenant diverses informations caractérisant le bloc. On y trouve notamment les champs `timestamp` correspondant à l'horodatage de la création du bloc, `parentHash` résultant de l'application de la fonction de hachage cryptographique Keccak-256 (Pritzker & May, 2015) (le prédécesseur du SHA-3) sur l'en-tête du précédent bloc, et le champ `transactionsRoot` contenant le condensat Keccak-256 des transactions présentes dans le corps du bloc (plus de détails sur ce point dans les sections suivantes).

Ici, ce sont les champs `parentHash` et `transactionsRoot` qui assurent l'inaltérabilité de la blockchain. En effet, si quelqu'un venait à remplacer un bloc par un autre, la valeur de hachage par Keccak-256 de son en-tête serait également modifiée (les chances de collisions étant extrêmement faibles). En comparant cette valeur avec le `parentHash` du bloc suivant, la tentative de remplacement peut donc être identifiée et rejetée. De plus, si un attaquant venait à modifier une ou plusieurs transactions d'un bloc, alors son `transactionsRoot` serait également changé. Étant présent dans l'en-tête du bloc, la modification de ce champ entraînerait également une différence entre le `parentHash` inscrit dans le bloc suivant et celui effectivement calculé, permettant donc d'identifier et de rejeter la modification frauduleuse.

3.3 TRANSACTIONS ET FONCTIONS DE TRANSITION D'ÉTATS

Le corps d'un bloc est constitué de transactions. Ce sont elles qui forment la base même des blockchains, permettant notamment aux utilisateurs de s'échanger des fonds.

3.3.1 TYPES DE COMPTES ETHEREUM

Une transaction est toujours réalisée d'un *compte* vers un autre (sauf dans le cas de la création d'un *smart contract*, où il n'y a pas de destinataire). Dans Ethereum, il y a deux types de comptes : les *externally owned accounts* (EOAs), contrôlés par leur clé privée respective, et les *contract accounts*, contrôlés par le code du *smart contract* correspondant (voir Section 3.5). Un compte est désigné par une *adresse*, soit un identifiant unique de 160 bits.

Les EOAs sont les comptes assignés à tout utilisateur quelconque, qu'il s'agisse du compte d'une personne réelle ou de celui d'un objet connecté qui interagirait avec la blockchain. Pour chaque EOA, une paire de clés privée-publique est générée aléatoirement. L'actuelle implémentation d'Ethereum utilise une cryptographie asymétrique par courbe elliptique, plus particulièrement l'algorithme ECDSA (Johnson *et al.*, 2001), qui utilise des clés publique et privée de 256 bits chacune. Dans le cas des EOAs, l'adresse d'un compte correspond aux 160 derniers bits de sa clé publique. Même si les clés sont générées localement et qu'aucun mécanisme de prévention de doublons n'existe, les chances que deux utilisateurs possèdent la même adresse sont en fait *dérisoirement* faibles, étant données les *quelques* 1.46×10^{48} possibilités dues aux 160 bits composants les adresses.

Les *contract accounts*, quant à eux, n'ont pas de paire de clés publique-privée. Leur adresse est plutôt déterminée par le hachage de la clé publique du EOA ayant déployé le contrat et du nombre de contrats qu'il a déjà créés jusqu'à présent (Buterin, 2016).

3.3.2 ÉTAT DE LA BLOCKCHAIN ETHEREUM

Les informations des comptes sont stockées dans une structure de données appelée l'*état* (*world state*, ou tout simplement *state* de la blockchain). Il est stocké dans une base de données propre à chaque client, en parallèle de la blockchain. Par exemple, l'implémentation officielle

d'Ethereum en langage Go, go-ethereum [19], utilise LevelDB, un gestionnaire de base de données utilisant un stockage sous la forme de paires clé-valeur et développé par Google. Ce format de stockage est particulièrement adapté à celui de l'état, qui est sauvegardé sous la forme d'une version modifiée de *Merkle Patricia trie* (Wood, 2019), résultat de la combinaison entre les arbres de Merkle (Merkle, 1982) et les arbres PATRICIA (ou arbre radix) (Morrison, 1968) ; dans ce cas chaque nœud est indexé par une valeur de hachage. Il est important de noter que l'état est bien distinct de la blockchain : celle-ci est persistante, et partagée par les clients du réseau blockchain, tandis que l'état est recalculé par les clients à chaque nouvelle transaction.

Format et stockage mis à part, l'état est en réalité un tableau associatif (*map*) qui à chaque adresse associe le compte correspondant, représenté par les champs suivants :

— `nonce` : dans le cas des EOAs, le nonce est égal au nombre de transactions envoyées par ce compte. S'il s'agit d'un *contract account*, celui-là est égal au nombre de contrats créés par le compte à l'origine de ce contrat.

— `balance` : le solde du compte, exprimé en Wei. Un Wei est la plus petite *coupure* de l'Ether (ETH), la devise utilisée dans Ethereum : 1 Wei = 10^{-18} ETH.

— `storageRoot` : dans Ethereum, un *contract account* a son propre espace de stockage persistant. Son contenu est sauvegardé en base de données sous la forme d'un *Merkle Patricia trie*, à l'instar de l'état de la blockchain.

— `codeHash` : dans le cas d'un *contract account*, il s'agit du code source haché du contrat correspondant.

19. https://github.com/ethereum/go-ethereum

3.3.3 TRANSACTIONS

L'état global de la blockchain est mis-à-jour localement à chaque transaction. En ce sens, les transactions servent à appliquer des fonctions de transition d'états (*state transition functions*).

Définition 6. *D'après Wood (2019), une fonction de transition d'état* Υ *est une fonction permettant de passer d'un état* σ *à un état* σ' *par l'application d'une transaction* valide T *:*

$$\sigma' = \Upsilon(\sigma, T)$$

Une transaction est valide à partir du moment où elle est correctement authentifiée et que son émetteur a les fonds suffisant pour l'exécuter. Par abus de langage et simplification, on dit aussi que, pour un état donné, une transaction est une fonction de transition d'état.

Concrètement, une transaction est une structure composée de plusieurs champs, dont `from`, l'adresse de l'émetteur de la transaction, `to`, l'adresse du destinataire, et `value`, le montant à transférer en Wei. Chaque transaction est hachée, sa valeur de hachage inscrite dans le champ `hash`, puis celle-ci est *signée* par l'émetteur via l'algorithme ECDSA, dont le résultat est renseigné dans les champs `r` et `s`. Cela signifie qu'en utilisant la clé publique de l'émetteur, le destinataire sera capable de déterminer si c'est bien la clé privée de celui-là qui a été utilisée pour générer la signature, lui permettant ainsi d'*authentifier* la transaction.

Cependant, au lieu de transmettre directement la clé publique dans la transaction comme ce pourrait être le cas, Ethereum (et d'autres blockchains telles que Bitcoin) utilise une méthode de récupération de la clé publique à partir de la signature elle-même (Brown, 2009; Wood, 2019; Breitner & Heninger, 2019) : avec les champs `r` et `s` de la transaction (en plus du champ `v` qui permet justement d'identifier la clé utilisée, d'après Wood (2019)), il est

possible de déterminer la clé publique utilisée pour la signature, d'en isoler l'adresse et de comparer celle-ci avec celle inscrite dans le champ `from`. Ainsi, si les deux adresses sont identiques, la transaction est bien authentifiée. Cette méthode, bien que nécessitant des calculs supplémentaires, permet de diminuer la taille des transactions en gardant la signature sans la clé publique.

Il est important de noter que la composition donnée ci-dessus diffère quelque peu de celle fournie par Wood (2019) dans le livre jaune d'Ethereum. En effet, dans ce dernier, les transactions sont *brutes* (*raw transactions*), tandis que dans ce chapitre nous considérons les transactions telles qu'elles sont représentées dans la blockchain même (après signature et inclusion dans un bloc). Le format sur lequel nous nous basons est celui du client Geth [20], implémentation officielle du protocole Ethereum, lorsqu'une requête est faite sur la valeur de hachage d'une transaction, comme rapporté dans la Figure 3.3 (certaines valeurs sont abrégées pour la lecture). Certains des autres champs de cette structure, notamment `gas`, `gasPrice`, et `input`, seront décrits dans la Section 3.5, dédiée aux *smart contracts*.

3.4 RÉSEAUX DE BLOCKCHAIN

Jusqu'ici, nous avons décrit la structure d'une blockchain et montré comment elle permettait d'avoir un registre immuable de transactions authentifiées. Toutefois, l'un des éléments clés de la sécurité des blockchains est qu'elles reposent sur des architectures de réseau pair-à-pair (*peer-to-peer*, ou plus simplement *P2P*), permettant de contourner les risques de *single point of failure* (c.-à-d. point individuel de défaillance, ou SPOF), problème récurrent des architectures client-serveur.

20. https://geth.ethereum.org/downloads/

```
$ eth.getTransaction("0xfb3283f693041fb2d21da2875c1c3670ed7...")

{
  blockHash: "0x7e38a795f1e4a12c0a640b3d60ffc05f0685d57d13f770013b4fba...",
  blockNumber: 2,
  from: "0x48f1d70fc2dfe9e2ed708567786e2a26ad492ba0",
  gas: 90000,
  gasPrice: 1,
  hash: "0xfb3283f6930418adb85dac47ad0514c2fb2d21da2875c1c3670ed7...",
  input: "0x",
  nonce: 1,
  r: "0xecd608010833543078 65fb15fdfbcff7854d059f63de667643d35a...",
  s: "0x128df31d492fc68a1fbe7e14b5559c1a82d96272fc7df65c71698c...",
  to: "0x4fd8ad74c2dfe9e2ed708567786e2a26ad49fde5",
  transactionIndex: 0,
  v: "0xa96",
  value: 100000000000
}
```

FIGURE 3.3 : Composition d'une requête Ethereum avec le client Geth © Quentin Betti.

Cependant, une telle structure rend la validation et l'insertion des transactions bien plus complexe. Dans un système client-serveur classique, le serveur serait chargé de traiter dans l'ordre les différentes transactions reçues, en s'assurant de leur validité puis en les agrégeant dans un bloc. Dans un réseau P2P, traiter les transactions dans l'ordre n'a plus réellement de sens car elles peuvent être générées en même temps un peu partout dans le réseau, et chacun de ses nœuds recevrait potentiellement les transactions dans des ordres différents, compromettant le principe même de la blockchain, avec sa construction par références inter-blocs.

Dans cette section, nous allons donc détailler le fonctionnement des réseaux de blockchains et décrire leur méthode pour établir une *version unique de la vérité* (*single version of truth*), principe fondamental des blockchains assurant que les utilisateurs disposent du même historique de transactions valides.

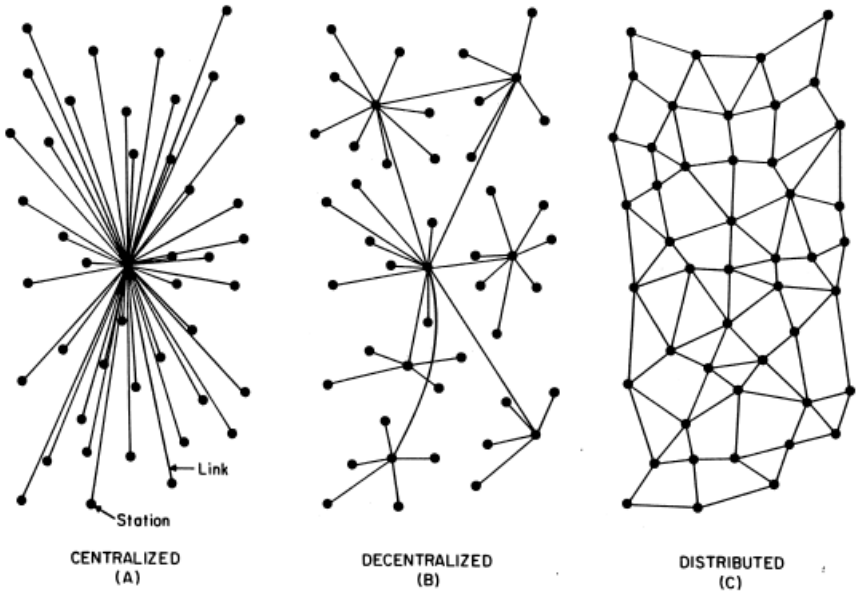


FIGURE 3.4 : Les différentes topologies de réseaux par Baran (1964) : centralisée (A), décentralisée (B), et distribuée (C) © 1964 IEEE.

3.4.1 TOPOLOGIE DISTRIBUÉE

Les réseaux de communication peuvent présenter trois types de topologies différentes (Baran, 1964) : centralisée, décentralisée, et distribuée. Celles-ci sont schématisées dans la Figure 3.4.

Une architecture centralisée est un système où des clients envoient des requêtes à un serveur. Dans ce contexte, celui-ci est l'unique autorité décisionnelle du traitement d'une requête et de l'envoi de la réponse au client. Bien qu'offrant un processus de traitement simple, un tel système présente de nombreux risques. En effet, puisqu'il offre un unique point d'entrée aux requêtes et présente ainsi un *single point of failure*, il est entre autres particulièrement vulnérable aux attaques par déni de service distribué (*Distributed Denial of Service attacks*, ou DDOS) et de l'homme du milieu (*man-in-the-middle attacks*). De plus, la compromission du

serveur peut avoir pour conséquence plusieurs effets clairement indésirables comme la fuite de renseignement personnelles ou la suppression de données d'affaires.

Les topologies décentralisées sont à mi-chemin entre les topologies centralisées et distribuées : les clients vont être répartis en groupes dans lesquels ils communiquent avec un unique serveur respectif. Ce dernier est également en communication avec un ou plusieurs autres serveurs de groupe. On parle de réseaux "décentralisés" car l'ensemble des clients ne repose pas nécessairement sur un unique point. En termes de sécurité, un tel réseau ne possède pas un *single point of failure*, mais plutôt un nombre fini de points de défaillance. Pris individuellement, chaque serveur est effectivement sujet aux mêmes risques d'attaques qu'un réseau centralisé. Dans l'ensemble, néanmoins, la compromission d'une communication ou d'un serveur n'impacte pas nécessairement la totalité du réseau, et peut permettre un fonctionnement normal pour la majorité des clients.

Très différente des topologies sus-mentionnées, les réseaux de blockchain présentent une architecture distribuée. Dans ce cas, plutôt que d'avoir un serveur répondant à une requête, tous les nœuds du réseau travaillent de façon collaborative sur une tâche. Les requêtes prennent alors la forme de messages qui sont propagés à travers le réseau par les nœuds. Plus précisément, les blockchains reposent sur des réseaux P2P, une sous-catégorie des topologies distribuées où tous les nœuds ont le même rôle, chacun étant à la fois *demandeur* et *fournisseur* de services (Schollmeier, 2001). En essence, chaque nœud possède une copie de la blockchain, est capable de recevoir des blocs, d'émettre des transactions et de propager celles d'autres nœuds. La seule exception concerne les *mineurs*, les nœuds chargés de former et valider les blocs de transactions en respectant un consensus pré-établi, puisqu'ils fournissent un travail supplémentaire par rapport aux autres. Néanmoins, n'importe quel nœud peut commencer ou arrêter de miner des blocs comme il le souhaite, rendant tous les nœuds du réseau bel et bien égaux. De plus, les mineurs ne sont en aucun cas des autorités du réseau, puisqu'ils fournissent

systématiquement les preuves de l'intégrité de leur travail, qui peut donc être vérifié et rejeté si nécessaire.

3.4.2 CYCLE DE VIE D'UNE TRANSACTION

Pour comprendre comment cette architecture P2P est capable d'accorder tous les nœuds sur une même version de blockchain, nous allons décrire le cycle de vie d'une transaction, de sa création à son ajout dans un bloc validé de la blockchain, en nous appuyant sur les travaux d'Infante (2018, 2019). Ce cycle de vie, schématisé dans la Figure 3.5, peut-être décomposé en cinq étapes.

Étape 1 : un client crée une transaction, la signe avec sa clé privée, puis l'envoie à un nœud Ethereum local. En réalité, le nœud n'est pas strictement local. En effet, celui-ci accepte des requêtes (via une interface JSON-RPC en l'occurrence) venant de programmes clients (Geth par exemple) de façon contrôlée. Pour éviter que le nœud reçoive des requêtes de clients inconnus, celui-ci est soit limité aux connexions via IPC (communication inter-processus) locales, ou via HTTP mais seulement pour un ensemble prédéfini d'adresses IP. C'est le cas des applications porte-monnaie (*wallet applications*), qui permettent aux utilisateurs de gérer leurs Ethers. Ces dernières peuvent communiquer avec un nœud qui est exécuté soit sur le même périphérique, soit sur un serveur distant autorisant les connexions venant du client.

Étape 2 : le nœud local reçoit la transaction signée, vérifie qu'elle est valide localement (c.-à-d. que la signature est correcte et que les fonds dépensés sont bien disponibles par rapport à l'état courant de la blockchain), et la transmet à ses nœuds voisins.

Étape 3 : la transaction est ensuite propagée dans le réseau par les différents nœuds. Par défaut, un nœud peut se connecter à 25 pairs différents.

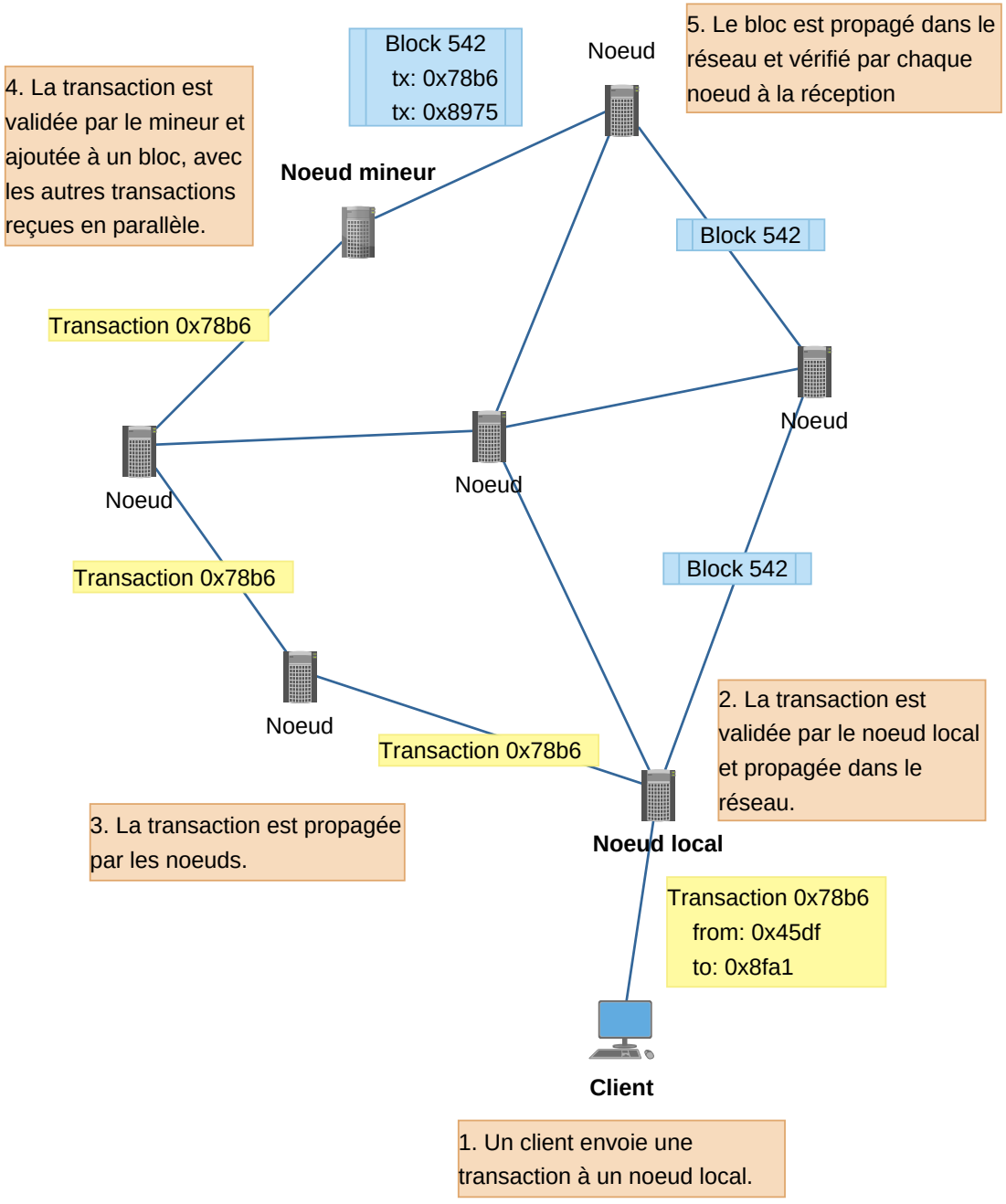


FIGURE 3.5 : Cycle de vie d'une transaction dans un réseau Ethereum simplifié © Quentin Betti.

Étape 4 : quand la transaction atteint un mineur, celui-ci va la valider et l'insérer dans un nouveau bloc avec les potentielles autres transactions qui lui sont parvenues entre temps. La preuve du respect du consensus de création des blocs, sur lequel les nœuds se sont mis d'accord au préalable, est alors intégrée à celui-là. Le mineur transmet ensuite ce bloc à ses pairs voisins.

Étape 5 : les nœuds recevant le nouveau bloc vont vérifier sa preuve de validité et l'ajouter à leur copie de la blockchain, mettant à jour son état local. Chaque nœud va alors propager le bloc à ses voisins.

De plus, le protocole prévoit le scénario où des blocs seraient minés en même temps par des mineurs différents, ceux-là pourraient alors contenir des transactions identiques et provoquer une incohérence dans la blockchain. À la réception de ces blocs, qui ont donc le même bloc parent, un nœud doit alors décider lequel ajouter dans sa blockchain. La méthode pour choisir le bloc est définie dans le consensus préalablement établi dans le réseau. Par exemple, dans le cas du consensus *Proof-of-Work* (voir Section 3.4.3), il est prévu, de façon arbitraire, que le bloc valide sera toujours celui avec la preuve de travail la plus élevée. Les autres blocs sont écartés : on parle alors de blocs *orphelins* (*orphan blocs*, dans le cas de Bitcoin) ou blocs *oncles* (*uncle blocs*, dans le cas d'Ethereum).

Finalement, pour des raisons de délai de propagation, si des nœuds continuent à miner des blocs en parallèle, formant ainsi des *branches* distinctes de la blockchain, ces deux branches seront acceptées, et les blocs continueront à s'ajouter. Cependant, après un certains temps ce sera la branche avec le plus grand nombre de blocs qui sera gardée, annulant les transactions de l'autre branche et les remettant en attente de validation.

3.4.3 CONSENSUS

Pour qu'un nouveau bloc de transactions soit accepté par les nœuds du réseau, il faut qu'il contienne la preuve que le mineur l'ayant créé ait respecté le consensus de génération des blocs, qui est établi auparavant à la mise en place du réseau. Chaque nœud est donc au courant du consensus utilisé et est capable de vérifier la preuve contenue dans les blocs. De manière générale, les consensus sont des mécanismes *Byzantine-Fault Tolerant* (BFT) : le terme fait référence au problème des généraux Byzantins de Lamport *et al.* (1982) où, sommairement, des généraux doivent se coordonner pour ordonner une attaque en s'envoyant des messagers, en supposant que des traîtres peuvent se cacher dans leurs rangs. Par extension, le caractère BFT d'un système assure donc son bon fonctionnement malgré la présence d'acteurs malveillants ou défaillants. Il devient donc évident que les consensus doivent être BFT, étant donné que les blockchains reposent sur des réseaux ouverts, acceptant tout nœud (potentiellement malicieux) souhaitant s'y connecter. Il existe de nombreux types de consensus, mais nous nous concentrerons particulièrement sur trois d'entre eux, adoptés par des réseaux majeurs de blockchain.

PROOF-OF-WORK

Le principe du consensus par preuve de travail, *Proof-of-Work* (PoW), réside dans un problème calculatoire qui ne peut être résolu que par force brute. Pour ce faire, Ethereum utilise l'algorithme Ethash (Ethereum Foundation, 2019a), similaire à Hashcash (Back, 2002), utilisé par Bitcoin. Son fonctionnement, simplifié, peut se résumer au pseudo-code présenté dans l'Algorithme 3.2.

Données : $header, difficulty$

Résultat : $nonce$

1 $target = compute\_target(difficulty)$

2 $nonce = random\_integer()$

3 **tant que** $hash(header, nonce) > target$ **faire**

4 $\quad nonce = nonce + 1$

5 **fin**

Algorithme 3.2 : Algorithme simplifié d'Ethash, adapté de Ethereum Foundation (2019a).

Une valeur cible, ou *seuil*, est tout d'abord calculée à partir d'une difficulté définie dans le bloc genesis et qui augmente au fil des blocs (ligne 2). Un nonce est ensuite aléatoirement choisi (ligne 3), et si le hachage combiné du nonce et de la valeur de hachage de l'en-tête du bloc est supérieur au seuil, alors on incrémente le nonce et on répète l'opération (ligne 4–6). Lorsque le résultat est inférieur ou égal au seuil, alors le nonce devient la preuve de travail (ligne 7) et est intégré dans le bloc par le mineur.

L'intérêt d'un tel algorithme est que ses besoins calculatoires sont *asymétriques*. En effet, le mineur n'a pas d'autre choix que d'itérer sur un très grand nombre de nonces avant de trouver une solution, étant donné le caractère quasi-imprédictible du résultat d'une fonction de hachage cryptographique. Le nombre de hachages par seconde que peut effectuer un mineur, appelé le *hashrate*, est d'ailleurs la métrique permettant de caractériser ses capacités de minage. Pour le nœud recevant un bloc miné, il est au contraire très facile d'en vérifier la preuve, puisqu'il suffit de hacher le nonce donné et l'en-tête du bloc, puis de comparer ce résultat avec le seuil.

Le caractère aléatoire de l'algorithme permet de s'assurer que la répartition des minages de blocs réussis soit homogène parmi la population des mineurs, à *hashrate* égaux. Pour être précis, pour qu'un mineur puisse valider un bloc (presque) à coup sûr, il faudrait qu'il possède la majorité de la puissance calculatoire du réseau afin d'être assuré d'être le plus rapide à miner les blocs. Dans ce cas-ci, qui reste théorique étant donné la puissance qu'il requiert, le mineur pourrait alors effectuer une attaque dite *des 51%* (Frankenfield, 2019), qui lui permettrait de centraliser la génération des blocs et donc de contrôler la blockchain, rendant son principe caduc.

Ce consensus permet donc une décentralisation de la génération des blocs, et récompense les minages réussis en accordant de la cryptomonnaie aux mineurs correspondant. Cependant, cette incitation financière pousse certaines entreprises à mettre en place des amas importants de mineurs. Pour le Bitcoin, ceci a pour conséquence qu'une dizaine de *mining pools* (regroupements de mineurs) se partagent à eux-seuls la quasi-totalité des blocs minés (BTC.com, 2019), remettant quelque peu en question le caractère décentralisé de la blockchain.

De plus, la consommation énergétique du PoW est également le sujet de nombreux débats (Vranken, 2017), certains pointant le fait qu'elle est comparable à celle de « petits » pays (O'Dwyer & Malone, 2014; Deetman, 2016; Quinn, 2017; Coppock, 2017; Beall, 2017), et d'autres insistant sur son insignifiance par rapport aux systèmes bancaires existants (McCook, 2014; Domingo, 2017; Canellis, 2018).

PROOF-OF-AUTHORITY

Le consensus par preuve d'autorité, *Proof-of-Authority* (PoA), est un mécanisme basé sur l'*identité* des nœuds. Contrairement à PoW, où tout nœud peut participer au minage des blocs, dans PoA la validation des blocs est laissée à la discrétion d'un ensemble de N nœuds,

qui sont les *autorités* du réseau (Angelis *et al.*, 2018). Ces autorités ont chacune un identifiant propre, et au moins $N/2+1$ (la majorité) d'entre elles sont présumées fiables. Les mécanismes de PoA reposent sur un système de *roulement* : pour un moment donné, une autorité, appelée alors *leader*, est habilitée à proposer un bloc.

Pour l'algorithme *Aura* (Parity Technologies, 2019), utilisé dans le client Ethereum Parity [21], le *leader* est déterminé par l'*étape* actuelle s de l'algorithme, définie par l'heure Unix t (*UNIX time*) sur laquelle les nœuds sont censés être synchronisés. L'identifiant l du *leader* est alors $l = s \pmod{N}$, où $s = t/step\_duration$ et $step\_duration$ est une constante définie à la mise en place du réseau.

Dans l'algorithme PoA utilisé par le client Geth, *Clique* (Szilágyi, 2017), le *leader* est déterminé en se basant sur le numéro du bloc courant et le nombre actuel d'autorités. De plus, d'autres autorités ont également le droit de proposer un bloc, à raison d'une fréquence de $N/2+1$ blocs.

Ce type de consensus est régi par le principe de *votes* : si un nœud propose un bloc qui n'est pas accepté par les autres autorités (*p. ex.*, bloc invalide ou proposé à la mauvaise étape), celles-ci votent pour l'évincer, lui retirant ainsi son statut d'autorité. Le fait qu'au moins $N/2+1$ autorités soient fiables permet à coup sûr d'évincer un *leader* malicieux par vote. Si elles ne le sont pas, alors le principe même du consensus est caduc.

L'avantage de PoA est qu'il requiert très peu de puissance calculatoire comparé au PoW, puisqu'il repose uniquement sur la vérification de signature et d'identité. Néanmoins, il est davantage vulnérable, puisqu'il suffit de corrompre la majorité des autorités (et non des nœuds, comme c'est le cas de l'attaque des 51% pour PoW) pour prendre le contrôle de la création des blocs.

21. https ://www.parity.io/

PROOF-OF-STAKE

La preuve d'enjeu, *Proof-of-Stake* (PoS) (Ethereum Foundation, 2017), est une famille de consensus où la validation de bloc dépend de l'*enjeu* économique proposé par le validateur. Il existe un parallèle entre PoS et PoA : dans PoA, les autorités *mettent en jeu* leur identité lorsqu'elles proposent un bloc ; dans PoS, les validateurs risquent leur cryptomonnaie. Un nœud souhaitant valider un bloc doit d'abord envoyer un type spécial de transaction qui verrouille un montant de cryptomonnaie décidé par le nœud, et qui est comparable à une caution. Plus le montant de cette caution est élevé, plus le nœud a de chances d'être nommé comme validateur du prochain bloc. Pour éviter que ce soit toujours le compte le plus riche qui soit élu validateur du prochain bloc, impliquant de fait une centralisation de la validation des blocs, il existe plusieurs méthodes adoptées par différentes implémentations, comme la pré-sélection pseudo-aléatoire de l'ensemble des potentiels validateurs, ou la mise en place de *tours de vote* entre ceux-ci pour choisir le bloc qui sera ajouté à la blockchain. Si le bloc proposé par un validateur est invalide (*p. ex.*, il contient une transaction frauduleuse), alors celui-ci perd l'argent mis en jeu dans la caution.

À l'instar de PoA, la puissance calculatoire demandée par PoS est bien en deçà de celle nécessaire pour PoW. De plus, il est considéré plus sécurisé que PoA, puisqu'il invite tous les nœuds à participer au consensus (ce qui est abordable, contrairement à PoW où la puissance de calcul nécessaire pour miner un bloc est très importante). Aussi, il n'est pas sensible à une attaque des 51%, puisque la puissance calculatoire n'est pas un facteur décisif du consensus. Pour contrôler la validation des blocs, il faudrait plutôt que l'attaquant possède 51% de la monnaie circulant sur le réseau (et qu'il soit prêt à la mettre en jeu), un scénario aussi peu probable que celui de l'attaque des 51%, et la répartition aléatoire des validateurs rend celui-là encore moins envisageable.

3.4.4 PERFORMANCES

L'un des principaux problèmes des plateformes telles que Ethereum et Bitcoin est qu'elles présentent des taux de transactions par seconde (TPS) *très* faibles : quelques dizaines de TPS au mieux sur les réseaux principaux. Dans les communautés de cryptomonnaies, il est courant de comparer le TPS d'un réseau de blockchain à celui de Visa, qui revendique la capacité de ses systèmes à supporter plus de 65.000 TPS (Visa, 2018). En pratique, Visa gère avec succès un taux moyen de 4.000 TPS, calculé à partir de leur 124.3 milliards de transactions traitées sur l'année 2018 (Visa, 2018). Comparé à ces chiffres, les réseaux Ethereum et Bitcoin sont donc beaucoup plus lents.

Cependant, d'autres réseaux de blockchain présentent des résultats prometteurs. Par exemple, EOS.IO est une plateforme de blockchain basée une architecture qui pourrait à terme supporter des millions de transactions par secondes, selon Block.one (2018). En pratique, EOS a récemment atteint son pic de vitesse avec près de 4 000 TPS (CryptoLions, 2019), ce qui reste tout de même loin des capacités maximales annoncées par Visa. Futurepia est une autre plateforme montrant également des résultats encourageants. Il s'agit un réseau blockchain en développement spécialisé dans les solutions pour plateformes type réseaux sociaux via des applications décentralisées (c.-à-d. les applications déployées sur les réseaux blockchain, aussi appelées DApps). Cette plateforme jouit d'une vitesse de 300.000 TPS selon Futurepia (2019b), mais ce chiffre reste tout de même à prendre avec précaution étant donné que les conditions d'évaluation (taille du réseau, type de transactions) restent floues.

Dans les faits, on constate toutefois de grandes différences entre les taux des réseaux blockchain majeurs et ceux de ces nouvelles plateformes. Pour cause, la plupart des premiers utilisent le consensus Proof-of-Work pour la génération des blocs qui, comme nous l'avons exposé plus tôt, est un mécanisme très couteux, à la fois en ressources et en temps. En l'oc-

currence, EOS et Futurepia utilisent des consensus basés sur Proof-of-Stake, respectivement *Delegated Proof-of-Stake*, ou DPoS (Bitshares, 2019), et *Dual Delegated Proof-of-Stake*, ou DDPoS (Futurepia, 2019a), qui sont bien plus rapides et économes en énergie que PoW.

3.5 SMART CONTRACTS

L'un des intérêts de la plateforme Ethereum est qu'elle est la première à avoir rendu possible le déploiement de *smart contracts* sur la blockchain. Théorisés par Nick Szabo à la fin des années 90 (Szabo, 1996, 1997), les *smart contracts* sont, à l'origine, des ensembles d'engagements ou promesses, spécifiés numériquement, comprenant les protocoles devant être utilisés par les différents partis pour interagir avec ces promesses (Szabo, 1996). Tel qu'implémenté dans Ethereum, un *smart contract* est du code informatique contenant une interface dont les services peuvent être appelés par des transactions envoyées à l'adresse de celui-là. Les transactions nouvellement reçues vont ensuite déclencher l'exécution du code correspondant par chaque nœud du réseau blockchain à travers l'EVM (*Ethereum Virtual Machine*), la machine virtuelle intégrée dans les clients Ethereum par laquelle passent toutes les transactions avant d'être ajoutées à la blockchain.

À l'instar des EOAs (c.-à-d. les comptes d'utilisateurs classiques, voir la Section 3.3.1), les *contract accounts* sont inscrits dans l'état global de la blockchain (voir la Section 3.3.2). En l'occurrence, dans celui-ci le champ `storageRoot` d'un tel compte correspond à la racine de l'arbre stocké en base de données (locale) représentant le contenu du *smart contract*. Il peut donc, par exemple, stocker des variables de façon persistante. Cependant, il existe bel et bien une limite théorique quant à la taille de la mémoire persistante d'un *smart contract*. En effet, l'état global de la blockchain est stocké sous la forme de paires clé-valeur, chaque clé et valeur ayant respectivement une taille de 256 bits et 32 octets (Wood, 2019) ; en d'autres mots, il existe 2^{256} clés possibles pouvant stocker 32 octets chacune. La taille maximale de

la mémoire persistante d'un contrat est donc de $2^{256} \times 32 = 2^{261}$ octets, soit un peu plus de $3,7 \times 10^{66}$ To. Étant donné que les valeurs de hachage des blocs ont une taille de 256 bits, cela veut dire qu'en pratique il est bien plus probable qu'une *collision* intervienne entre des blocs avant qu'un contrat atteigne sa taille de stockage maximale.

3.5.1 SPÉCIFICATION DES SMART CONTRACTS

Les *smart contracts* Ethereum sont écrits en Solidity (Ethereum Foundation, 2019b), un langage haut niveau orienté objet. Un *smart contract* est donc composé de champs et de méthodes définis par l'utilisateur. Pour illustrer cela, prenons l'exemple simple d'un contrat appelé `MyStringStorageContract`, dont le but est de stocker une chaîne de caractères (*string*) en mémoire et qui permet aux utilisateurs de modifier celle-ci. Un tel contrat peut être spécifié avec la Figure 3.6.

Le contrat est instancié en passant une chaîne de caractères au constructeur qui initialise la valeur de celle en mémoire (lignes 12–14). Une méthode `getMyString` est fournie afin de permettre aux utilisateurs de récupérer la valeur de la chaîne (lignes 16–19). La méthode `setMyString` permet quant à elle de modifier la chaîne et, à chaque appel, elle émet un *log event* `OnStringChange` contenant l'ancienne et la nouvelle valeur de la chaîne de caractères (lignes 21–25).

Les *log events* sont des événements qui, lorsqu'ils sont émis, sont inscrits dans le champ `logs` de la transaction qui les a provoqués. La visualisation d'un *log event* `OnStringChange` dans la transaction est donnée dans la Figure 3.7. On peut y identifier l'adresse du *smart contract* émetteur (champ `from`), le nom de l'événement (champ `event`) et ses arguments (champ `args`). Ici, on peut voir que la *string* "`Hello, world!`", avec laquelle nous avions initialisé le contrat, a été remplacée par "`Hello, you!`" (les arguments sont à la fois référen-

```
pragma solidity >=0.4.22 <0.6.0;

contract MyStringStorageContract {

    string private myString;

    event OnStringChange(
        string oldString,
        string newString
    );

    constructor (string memory _myString) public {
        myString = _myString;
    }

    function getMyString () external view
        returns (string memory _myString) {
        _myString = myString;
    }

    function setMyString (string memory _myNewString) public {
        string memory oldString = myString;
        myString = _myNewString;
        emit OnStringChange (oldString, myString);
    }
}
```

FIGURE 3.6 : Code Solidity du contrat MyStringStorageContract, de Betti *et al.* (2020). Reproduit avec la permission de Springer.

```
$ eth.getTransaction("0xc390a9f2ef9fee3a406994318dffab7b486...")

{
  ... // Autres champs
  logs: [{
    "from": "0x692a70d2e424a56d2...",
    "topic": "0x111001b6e910b4bc...",
    "event": "OnStringChange",
    "args": {
      "0": "Hello, world!",
      "1": "Hello, you!",
      "oldString": "Hello, world!",
      "newString": "Hello, you!",
      "length": 2
    }
  }]
}
```

FIGURE 3.7 : Exemple d'émission de *log event* OnStringChange © Quentin Betti.

cés par index et par clé, ce qui explique leur duplication). Le `topic` quant à lui est un champ particulier qui peut être utilisé pour indexer les arguments présents dans les événements et permettre une recherche plus rapide de ceux-ci à travers l'entièreté de la blockchain.

Actuellement, l'outil recommandé par *The Ethereum Foundation* pour rédiger des *smart contracts* est Remix [22], un IDE web qui permet également de compiler les *smart contracts*, de simuler des transactions vers ceux-ci et d'explorer leur résultat.

3.5.2 DÉPLOIEMENT ET TRANSACTIONS

Pour déployer un *smart contract* sur un réseau blockchain Ethereum, il faut tout d'abord le compiler afin d'obtenir son *bytecode*, c.-à-d. le code qui sera interprété par l'*Ethereum Virtual Machine* (EVM) (Wood, 2019). L'EVM est la machine virtuelle incluse dans les clients

22. https://remix.ethereum.org/

Ethereum qui permet d'exécuter les transactions, qu'il s'agisse de transferts d'Ether classiques ou d'appels aux méthodes de *smart contracts*. Toute transaction est d'abord exécutée dans l'EVM afin d'être validée et ajoutée dans un bloc. En pratique, le *bytecode* peut s'obtenir directement sur Remix, ou via d'autres outils comme solc, un compilateur Solidity écrit en Javascript.

BYTECODE ET *OPCODES*

Plus précisément, le *bytecode* est une représentation hexadécimale des *opcodes* du *smart contract*. Les *opcodes* sont les instructions de base comprises par l'EVM. Lorsque compilé, un contrat est donc traduit en une liste d'instructions que l'EVM pourra exécuter. Par exemple, la Figure 3.8 liste les dix premières instructions du *smart contract* `MyStringStorageContract`, décrit plus tôt dans la Figure 3.6. Pour chaque nom d'*opcode*, une valeur hexadécimale est attribuée (*p. ex.*, `PUSH1` est représentée par `0x60`, `MSTORE` par `0x52`). La liste complète des *opcodes*, de leur description et représentation hexadécimale est fournie en annexe du livre jaune d'Ethereum (Wood, 2019). Le *bytecode* correspondant à ces opérations (et donc au début du contrat `MyStringStorageContract`) est alors `0x608060405234801561001057600080`.

Pour déployer ce *smart contract* sur le réseau, il suffit alors d'envoyer une transaction sans destinataire, indiquant à l'EVM qu'il s'agit d'une création de contrat. De plus, le *bytecode* ainsi que le code d'initialisation (c.-à-d. appel du constructeur et renseignement de ses paramètres, s'il y en a) sont inscrits dans le champ `input` de la transaction, dont la valeur commencera donc par `0x608060405234801561001057600080` dans le cas du `MyStringStorageContract`.

22. https://www.npmjs.com/package/solc

```
1  PUSH1 0x80
2  PUSH1 0x40
3  MSTORE
4  CALLVALUE
5  DUP1
6  ISZERO
7  PUSH2 0x10
8  JUMPI
9  PUSH1 0x0
10 DUP1
```

FIGURE 3.8 : Les dix premiers *opcodes* du *smart contract* © Quentin Betti.

LIMITATION DES OPÉRATIONS

Comme nous l'avons dit précédemment, toute transaction incluse dans un bloc est exécutée par la totalité des nœuds du réseau pour mettre à jour leur propre état de la blockchain Ethereum. Dans un tel scénario, le réseau serait vulnérable à une surcharge de code à exécuter. Un utilisateur mal intentionné pourrait en effet envoyer une transaction provoquant l'exécution d'un code très gourmand en ressources et ainsi affecter les performances du réseau et des nœuds connectés.

Pour empêcher cela, Ethereum a introduit la notion de carburant, *gas* en anglais. Le *gas*, est aux transactions ce qu'est l'essence aux voitures : il représente la consommation d'une transaction en termes calculatoires. Pour chaque *opcode*, on associe un nombre d'unités de *gas* (*p. ex.*, `PUSH1` consomme 3 unités de *gas* par appel, tandis que `JUMPI` en nécessite 10), et lorsqu'une transaction est envoyée, l'EVM exécute un à un les *opcodes* et garde en mémoire la quantité de *gas* consommée en cumulé, appelée `gasUsed`. L'utilisateur envoyant la transaction, quant à lui, doit renseigner la quantité de *gas* maximale `gasLimit` qu'il souhaite allouer pour celle-ci. Si, à un moment de l'exécution, `gasUsed` surpasse `gasLimit`, alors celle-ci est arrêtée soudainement et la transaction est rejetée.

Le concept de *gasLimit* est aussi applicable aux blocs : une *gasLimit* par bloc est définie pour le réseau, et la somme du `gasUsed` des transactions d'un bloc ne peut être supérieure à cette limite (actuellement de 10 000 unités). Cela permet notamment de limiter le nombre de transactions qu'un seul et même bloc peut contenir.

De plus, comme l'essence, le *gas* a un prix (`gasPrice`). Celui-ci n'est pas fixe, mais plutôt défini par l'utilisateur pour une transaction spécifique. Il devra alors payer le *gas* dépensé à hauteur de `gasUsed` $\times$ `gasPrice`, ce qui peut s'apparenter à des frais de transactions. On serait alors tenté de penser que l'utilisateur appliquerait un prix de *gas* très bas, voire nul, pour chacune des transactions qu'il souhaite envoyer. Néanmoins, le choix d'accepter ou non une transaction avec un bas prix de *gas* est à la discrétion du mineur du bloc correspondant, pour la simple raison que les frais des transactions d'un bloc miné lui reviennent. Un mineur a donc tout intérêt à ce que le `gasPrice` d'une transaction soit élevé et, si le prix est trop bas à ses yeux, il n'intégrera pas la transaction au bloc. Celle-ci pourrait alors être intégrée à un bloc par un mineur moins regardant sur le `gasPrice`, ou ne jamais l'être. En ce sens, une transaction avec un haut `gasPrice` a plus de chance d'être incluse rapidement dans un bloc qu'une à bas `gasPrice`.

Il est également important de noter que, pour pouvoir effectuer une transaction, un compte doit nécessairement avoir des fonds supérieurs ou égaux à `gasLimit` $\times$ `gasUsed`, même si finalement ce sera bien un montant de `gasUsed` $\times$ `gasPrice` qui sera prélevé, en plus des fonds potentiellement impliqués dans la transaction elle-même bien évidemment.

Pour résumer, ce système empêche donc bien la consommation excessive de puissance calculatoire du réseau en forçant l'utilisateur à payer des frais de transactions proportionnels à la quantité d'instructions et à la « puissance » qu'elles requièrent. S'il ne peut s'acquitter de ces frais, la transaction est tout bonnement rejetée.

3.6 APPLICATIONS

Comme nous l'avons vu tout au long des sections précédentes, les blockchains sont des registres ouverts, où toute modification ou transaction illégitime est inévitablement détectée et rejetée par les pairs au sein du réseau. Pour ces raisons, elles sont considérées sécurisées et fiables. Ces caractéristiques sont particulièrement pertinentes dans le cadre des cryptomonnaies telles que le Bitcoin ou l'Ether. Néanmoins, ses champs d'applications n'ont cessé de croître depuis sa création, notamment grâce à la popularisation des *smart contracts*. Ici, nous présentons brièvement les domaines dans lesquels les notions de blockchain et de cycle de vie sont étroitement liées.

3.6.1 DOSSIERS MÉDICAUX ÉLECTRONIQUES

Le secteur de la santé est l'un des domaines où la blockchain est vue comme particulièrement prometteuse. Ce secteur fait intervenir plusieurs acteurs ; on peut citer, entres autres, les médecins de famille, les spécialistes, le personnel infirmier, les hôpitaux, les cliniques, les laboratoires, les pharmaciens et, bien sûr, les patients. Le domaine est aussi étroitement lié à d'autres secteurs comme celui des assurances pour la prise en charge des coûts de soin, et de la *supply chain* pour l'acheminement de matériel médical et l'approvisionnement en médicaments. On y observe donc de nombreux acteurs différents collaborant vers un même but : assurer l'accès aux soins appropriés pour les patients. Cela passe notamment par la collecte et le stockage de *données médicales*, qu'elles soient issues d'un examen en laboratoire ou d'une consultation médicale.

Plusieurs travaux (Hölbl *et al.*, 2018; Agbo *et al.*, 2019; Mettler, 2016) montrent que la gestion des *dossiers médicaux électroniques* (EMRs, pour *Electronic Medical Records*) est une application de la blockchain au domaine médical particulièrement pertinente. Ici, le

but est de faciliter l'accès aux EMRs par le patient et le personnel médical pour empêcher la *fragmentation* des données, c.-à-d. faire en sorte que chaque acteur puisse avoir accès aux informations qui lui sont nécessaires. Il faut néanmoins que le patient soit le *propriétaire* de ses données, qui doivent rester confidentielles autant que faire se peut, et qu'il puisse décider d'accorder l'accès à celles-ci comme il le souhaite.

Par exemple, Liang *et al.* (2017) montrent qu'il est possible d'atteindre ce but avec une approche semi-décentralisée. Dans ce cas, une blockchain basée sur *Hyperledger Fabric* (The Linux Foundation, 2020) sert à garantir l'intégrité des informations en stockant les condensats des données ajoutées aux dossiers dans des transactions. Elle propose également un système de gestion des accès en statuant sur l'octroi de l'accès aux données à un acteur de type assurance ou médical ; ces accès sont définis et modifiables par le patient. Les données en elles-mêmes sont quant à elle stockées sur des bases données classiques sécurisées. Cependant, ce dernier point présente potentiellement un *single point of failure*. Un piratage des serveurs contenant les bases de données pourrait mener à une fuite catastrophique d'informations confidentielles.

Azaria *et al.* (2016) ou encore Nchinda *et al.* (2019) proposent un système similaire, en se basant cette fois-ci sur un réseau Ethereum et l'implémentation de *smart contracts* pour gérer l'accès aux données. Celles-ci sont rendues disponibles via une base de données propre à chaque nœud du réseau (*p. ex.*, le nœud du laboratoire d'analyse stockera les résultats de ses propres examens dans sa base de données). Au lieu d'une seule base de données conservant l'intégralité des informations, celles-ci sont réparties sur les nœuds des différents services de soin. Sans adopter une architecture totalement décentralisée, cela réduit quand même le risque de fuite d'information (au lieu de l'intégralité des données du patient, seules celles du nœud piraté seraient divulguées).

Munoz *et al.* (2019) utilise également une blockchain de type *Hyperledger Fabric* avec des base de données classiques pour stocker les données des EMRs. Comme ils le font justement remarquer, les applications utilisant des systèmes externes en complément de la blockchain sont les plus répandues (ceci est d'ailleurs valable pour les domaines d'application suivants). Il y a deux principales raisons à cela : d'une part cela facilite l'adoption du nouveau système qui se base en partie sur ceux déjà en place, et d'autre part cela permet d'être en accord avec certaines directives sur les données personnelles, comme le RGPD en Europe (The European Parliament and the Council of the European Union, 2016), qui édicte le *droit à l'oubli* pour les utilisateurs. La sécurité de la blockchain étant en partie basée sur la persistance des données qu'elle contient, les informations en elles-mêmes doivent être stockées ailleurs pour qu'elles puissent potentiellement être supprimées ; la blockchain se contente de garder seulement les condensats des données pour en assurer leur intégrité.

3.6.2 INTERNET DES OBJETS

L'Internet des objets (ou IoT, pour *Internet of Things*) est défini comme un ensemble « d'objets », tels que des tags RFID, capteurs, actionneurs ou téléphones mobiles, capables d'interagir entre eux et de coopérer avec les systèmes à proximité pour atteindre un but commun (Giusto *et al.*, 2010). L'hétérogénéité et le nombre de ces objets imposent de nouveaux défis, incluant notamment la gestion de ces flottes massives d'objets, leur intercommunication et leur sécurité.

Khan & Salah (2018) ont d'ailleurs récemment publié une étude sur les problèmes rencontrés dans l'IoT et expliquent comment certains d'entre eux pourraient être résolus grâce à l'intervention de la blockchain. Ainsi, en stockant les données échangées par les objets dans une blockchain, leur intégrité et authenticité serait intégralement assurée par le fonctionnement même de la blockchain, limitant ainsi, par exemple, le risque d'usurpation

d'identité d'objets et la collecte de fausses données. De plus, les *smart contracts* permettraient d'établir facilement et dynamiquement les règles d'accès, en restreignant l'ensemble des systèmes autorisés à effectuer certaines actions grâce à leur adresse au sein de la blockchain. On pourrait d'ailleurs imaginer qu'à terme cette adresse serve de substitut aux adresses IPv4 et IPv6 standard qui présentent des espaces d'adressage de 32 bits et 128 bits respectivement, contre 160 bits pour les adresses Ethereum. Offrant une probabilité de collision d'adresses de 10^{-48}, cela pourrait être considéré suffisant pour fournir un identifiant unique à chaque objet sans nécessiter d'autorité centrale, à contrario de l'IPv4 et l'IPv6 où les blocs d'adresses sont attribués par l'IANA.

Concrètement, Dorri *et al.* (2017a) ont montré comment la blockchain pouvait gérer les objets connectés et les données qu'ils génèrent dans le cadre de *maisons intelligentes*. Ici, chaque maison possède son propre réseau de blockchain privé, dans lequel chacun des objets représente un nœud. Un *home miner* est également affecté à chaque réseau privé ; il s'agit d'un dispositif capable de traiter les transactions internes et externes à la maison (cette mission ne pouvant être assignée à *tous* les objets à cause de leurs limitations en termes de puissance calculatoire et de stockage) qui pourrait être, par exemple, directement intégré dans la passerelle du domicile. L'ajout ou le retrait d'objets connectés dans le réseau est géré dans la blockchain, en y ajoutant les transactions correspondantes.

Dorri *et al.* (2017b) proposent quant à eux une solution d'architecture blockchain pour les véhicules intelligents interconnectés. De façon semblable à l'exemple précédent des maisons intelligentes, chaque véhicule est équipé d'une interface sans fil qui le connecte à un réseau blockchain pouvant être composé, par exemple, des véhicules, des constructeurs automobiles, des fournisseurs de logiciels internes ou de stockage de données en ligne. On utilise un dispositif de stockage local pour sauvegarder les informations sensibles ou privées (telles que la position ou l'historique de maintenance du véhicule) ; leurs condensats sont

envoyés sous la forme de transactions au réseau. De cette façon, en cas de litige, l'utilisateur peut prouver la légitimité de ses informations en comparant le condensat de celles stockées localement et celui présent dans la blockchain.

3.6.3 GESTION DES PROCESSUS D'AFFAIRES

La gestion des processus d'affaires (BPM) est également un domaine d'application de technologies blockchains particulièrement prometteur. Le BPM vise à organiser les étapes et répartir les ressources (c.-à-d. construire et implémenter le cycle de vie) de tout processus de production ou de réalisation de service, en l'optimisant au mieux dans la plupart des cas. Pour une entreprise, cela revient à orchestrer les tâches de quelques dizaines à plusieurs centaines (voire milliers) d'employés et de collaborateurs externes. C'est dans ce cadre que l'utilisation de la blockchain présente de grands intérêts. Certes, les opérations intra-entreprise peuvent être considérées fiables (bien que des fraudes internes soient toujours possibles), mais celles réalisées en externe nécessitent une surveillance et un contrôle particulier : un prestataire pourrait, pour des raisons qui lui sont propres, falsifier des données pour, par exemple, mentir sur la provenance ou la méthode de fabrication d'un produit.

Plus largement, Mendling *et al.* (2018) ont identifié les apports potentiels mais également les manquements des technologies blockchains à tous les niveaux de l'implémentation du BPM. Par exemple, dans la phase de supervision (*monitoring*) de l'exécution du processus (qui consiste principalement à s'assurer que le cycle de vie effectif du processus correspond bien à celui spécifié), la blockchain permettrait à chacun des intervenants d'effectuer la vérification indépendamment des autres. Les auteurs notent cependant que cela nécessite de produire au préalable une *trace* complète des événements, certaines parties des données étant probablement stockées hors-blockchain et possiblement chiffrées.

Dans un autre contexte, Viriyasitavat *et al.* (2018) propose une solution pour automatiser la sélection de service au sein d'un processus métier en se basant particulièrement sur la *qualité de service* (QoS, pour *Quality of Service*). Ils proposent ainsi une plateforme appelée QoS blockchain. Cette dernière est basée sur un système de *smart contracts* responsables de collecter les indicateurs de QoS pour différents services et permet donc de se passer de certains tiers de confiance, nécessaires jusque-là pour recueillir et mettre à disposition les données de QoS. Ceci a pour directe conséquence une diminution des coûts associés à ces intermédiaires et leur fonctionnement, ainsi qu'une accélération des transactions par l'automatisation de la sélection et la composition des services.

López-Pintado *et al.* (2017) ont quant à eux développé Caterpillar, un système de BPM basé entièrement sur une blockchain Ethereum où les modèles de processus d'affaires sont spécifiés en BPMN. Ainsi, à partir d'un modèle BPMN fourni au format XML, un module est responsable de générer le code Solidity du *smart contract* correspondant. D'autres composants permettent ensuite d'interagir avec le modèle, *p. ex.* par la création d'une nouvelle instance du modèle et la récupération ou mise à jour de l'état d'un processus. Ces différentes fonctionnalités sont disponibles directement depuis une API REST fournie par le moteur de Caterpillar.

CHAPITRE IV
SUPERVISION D'UN SYSTÈME DE SUIVI DE COLIS BASÉ SUR UNE BLOCKCHAIN

Depuis 2016, la mise en pratique de la blockchain et des *smart contracts* dans les domaines de la logistique et de la chaîne d'approvisionnement (*supply chain*) est considérée comme particulièrement prometteuse (Wang *et al.*, 2019). Les nouveaux concepts de ce domaine, tels que l'Internet Physique et les logistiques hyperconnectées (décrits dans la Section 1.2), présentent néanmoins de nouveaux défis en termes de vérification du respect du cycle de vie des colis. En effet, le partage des ressources (physiques ou numériques), la multiplicité des acteurs participant à l'acheminement des paquets, la complexification des cycles de vie même des artefacts, et la décentralisation intrinsèque à ces modèles requièrent des technologies adaptées.

En conséquence, après une présentation de l'intérêt de l'utilisation des technologies blockchains dans le domaine de la *supply chain*, le premier but de ce chapitre est de montrer comment celles-ci peuvent être concrètement mises à profit pour créer un journal distribué, sans propriétaire et sécurisé, de tous les événements relatifs à l'entièreté d'une *supply chain* dans un contexte de logistique urbaine hyperconnectée. Nous verrons qu'un des avantages de cette configuration est qu'elle rend possible l'envoi de requêtes sur cette blockchain afin de vérifier des propriétés garantissant le fonctionnement correct et efficace de la *supply chain*. Puis, il s'agira de définir certaines de ces propriétés qui nous semblent pertinentes à superviser, et de les implémenter à l'aide de la bibliothèque de CEP BeepBeep, que nous avons présentée en Section 2.2.1.

4.1 BLOCKCHAIN ET SUPPLY CHAIN

Dès 2016, l'introduction de la blockchain dans la *supply chain* fut perçue comme une réelle opportunité dans ce domaine et a été le sujet de nombreux travaux depuis. En effet, dans un tel contexte, plusieurs acteurs doivent collaborer afin de satisfaire une demande, ce qui nécessite que ceux-ci puissent se faire confiance et, dans une certaine mesure, partager des ressources.

Prenons un exemple simple : un acheteur passe une commande en ligne à un vendeur pour un produit donné. Puisque le vendeur ne possède sans doute pas son propre site internet pour vendre ses produits, la commande sera probablement passée via un détaillant intermédiaire ou une place de marché en ligne de type Amazon ou eBay. Une fois le paiement effectué, l'intermédiaire doit informer le vendeur qu'une commande a été réalisée, et une fois celle-ci confirmée par ce dernier, il doit récupérer l'adresse de l'acheteur et expédier le produit via un service de livraison. À partir de cet instant, le colis va passer par différents centres de tri et entrepôts, transporté par des livreurs ou des camions de livraisons, jusqu'à sa destination finale, où l'acheteur confirmera la réception de la commande. Notons que, dans ce scénario, sont déjà présents six différents types d'acteurs : l'acheteur, le vendeur, la place de marché intermédiaire, le service de livraison, les livreurs, et les conducteurs de camions de livraison.

Bien évidemment, cet exemple est considérablement simplifié, et pourtant il ne serait pas aisé d'identifier les raisons de la perte d'un paquet ou du retard d'une livraison, entre autres. Le dernier livreur l'a-t-il bien transmis à l'acheteur, ou l'aurait-il gardé pour lui-même ? L'acheteur pourrait mentir, en affirmant ne pas avoir reçu le colis alors que c'est le cas, dans l'espoir d'en obtenir un second gratuitement. Ou peut-être qu'une défaillance dans le traitement du paquet s'est produite au niveau du service de livraison, et celui-ci l'aurait « oublié » au fond d'un entrepôt, ou envoyé à une mauvaise adresse. De fait, même dans ce simple exemple, identifier

les défaillances dans le transport de marchandises et trouver les solutions appropriées pourrait être long et complexe sans un système de suivi fiable et adéquat. Imaginez, alors, si nous avions affaires à un scénario de *supply chain* complet, où des pièces de produits viennent de différents fournisseurs et sont transportées par plusieurs services de livraison à travers de nombreux pays, avant d'être assemblées en produits qui seront exportés aux détaillants où, finalement, ils seront achetés ou livrés aux clients.

De plus, chacun des acteurs utilise probablement son propre système de suivi interne. Cela a pour conséquence que lorsque l'un d'eux traite un paquet, les autres sont complètement laissés dans l'ignorance et doivent se fier aveuglément aux données et informations fournies par ce dernier. Ceci laisse la porte ouverte aux vols, contrefaçons, ou falsifications de l'origine des produits, qui sont des sujets de préoccupation majeurs des entreprises et des consommateurs, particulièrement dans les industries du luxe et de l'agroalimentaire. Pour limiter ces risques, des tierces parties indépendantes (*p. ex.*, gouvernements, labels) sont souvent nécessaires, mais les procédures peuvent être longues, coûteuses, sans être entièrement à l'abri de risques, puisque la corruption de ces entités est toujours envisageable (Yardley & Barboza, 2008).

Contournant ces problèmes, la blockchain peut être utilisée pour permettre un système de suivi fiable et transparent (Wang *et al.*, 2019; Kshetri, 2018; Abeyratne & Monfared, 2016). En effet, comme nous l'avons mentionné plus tôt, la blockchain est un registre ouvert, décentralisé et immuable, où chaque transaction est authentifiée. En adoptant cette technologie dans un système de suivi commun aux acteurs, ceux-ci auraient alors une bien plus grande confiance envers les données qu'elle contient, puisqu'un attaquant ou acteur malveillant ne pourrait les modifier ou usurper l'identité de quelqu'un sans que cela soit détecté. Par exemple, Madhwal & Panfilov (2017) relèvent les bénéfices d'une telle technologie en l'appliquant à la gestion logistique de la production d'avions. En 2016, BigchainDB GmbH (2018) introduit BigchainDB, un système de base de données décentralisée basé sur la blockchain afin de

résoudre une partie des problèmes de scalabilité des blockchains standards. Cette technologie, avec l'aide de l'IoT, fut alors mise en application par Feng Tian (2017) dans un système de traçabilité alimentaire. Toyoda *et al.* (2017) ont illustré le potentiel d'Ethereum et des *smart contracts* pour mettre en place un système anti-contrefaçon dans la post-*supply chain*, en se basant sur des tags RFID et un standard global pour l'identification des articles commerciaux, le SGTIN-96. De plus, puisque l'utilisation de l'IoT est de plus en plus envisagée dans la chaîne logistique (Wang & Liu, 2014; Mo, 2011; Shanahan *et al.*, 2009), la blockchain peut même être déployée pour augmenter la sécurité des appareils IoT opérant dans ce contexte (Kshetri, 2017). À vrai dire, de nombreux travaux décrivent les intérêts et implémentations des technologies dans la gestion de la *supply chain*, et Wang *et al.* (2019) ont récemment réalisé une étude exhaustive de telles publications ou évolutions industrielles.

4.2 MISE EN PLACE DU SYSTÈME DE SUIVI

Notre système de suivi de colis s'appuie sur une simulation de logistique hyperconnectée développée par Kaboudvand *et al.* (2018). Cette simulation a été réalisée à l'aide d'AnyLogic [23], un environnement de développement de simulations en Java populaire, notamment dans le milieu de la *supply chain*. Après avoir présenté cette simulation, nous montrons comment nous pouvons l'adapter afin que toutes les actions réalisées sur les colis par les agents internes à la simulation AnyLogic soient sauvegardées directement dans la blockchain, au lieu d'une base de données ou de fichiers de journaux standards. Pour ce faire, nous établissons un réseau privé Ethereum auxquels les agents se connectent et envoient des informations sur les actions qu'ils effectuent via des *smart contracts*.

23. https ://www.anylogic.com/

4.2.1 UNE SIMULATION ANYLOGIC POUR LA LOGISTIQUE HYPERCONNECTÉE

Pour illustrer le potentiel de la logistique hyperconnectée sur l'efficacité de la *supply chain*, une simulation en milieu urbain a été développée par Kaboudvand *et al.* (2018) avec le logiciel AnyLogic et présentée à l'*IISE Annual Conference* de 2018. Dans cette simulation, les auteurs considèrent le modèle suivant d'une « mégalopole » simplifiée, en utilisant les concepts de logistique hyperconnectée définis en Section 1.2. La ville elle-même est schématisée par une carte rectangulaire et est limitée à une seule *area* (voir Figure 4.1) ; cette *area* est elle-même composée de quatre *local cells*. Des colis à livrer apparaissent dans la ville, et les transporteurs (*transporters*) ainsi que les livreurs (*deliverers*) participent à leur acheminement jusqu'à leur destination finale, à l'intérieur de la ville. Chaque *local cell* est divisée en neuf *unit zones*, dont la clarté représente la densité de demande : plus une *unit zone* est foncée, plus haute est la probabilité que celle-ci soit l'origine ou la destination d'un colis. Les *access hubs* sont placés à chaque sommet d'une *unit zone* et sont partagés par les *unit zones* adjacentes. Les *local hubs* sont quant à eux placés aux sommets des *local cells* et sont partagés par celles qui leur sont adjacentes. Dans ce scénario, tous les *local hubs* sont également des *access hubs*.

Trois types de moyens de transport sont définis dans ce modèle : les *couriers* (coursiers), les *riders* (conducteurs) et les *shuttlers* (conducteurs de navettes). Les *couriers* s'occupent des livraisons « du dernier kilomètre » au sein de leur *unit zone* assignée, à pied, à vélo, en scooter ou à bord d'un véhicule électrique léger. Les *riders* et les *shuttlers* utilisent des véhicules plus rapides (généralement électriques ou fonctionnant au gaz naturel), puisqu'ils parcourent de plus longues distances. Les *riders* sont assignés à une *local cell*, transportant les marchandises entre les *unit zones* et/ou les *local hubs* qui sont à l'intérieur de celle-là. Les *shuttlers* sont assignés à une *area*, transportant les marchandises entre les *local hubs* à l'intérieur de celle-ci, et/ou entre les *gateway hubs* (non représentés ici) permettant les acheminements vers d'autres

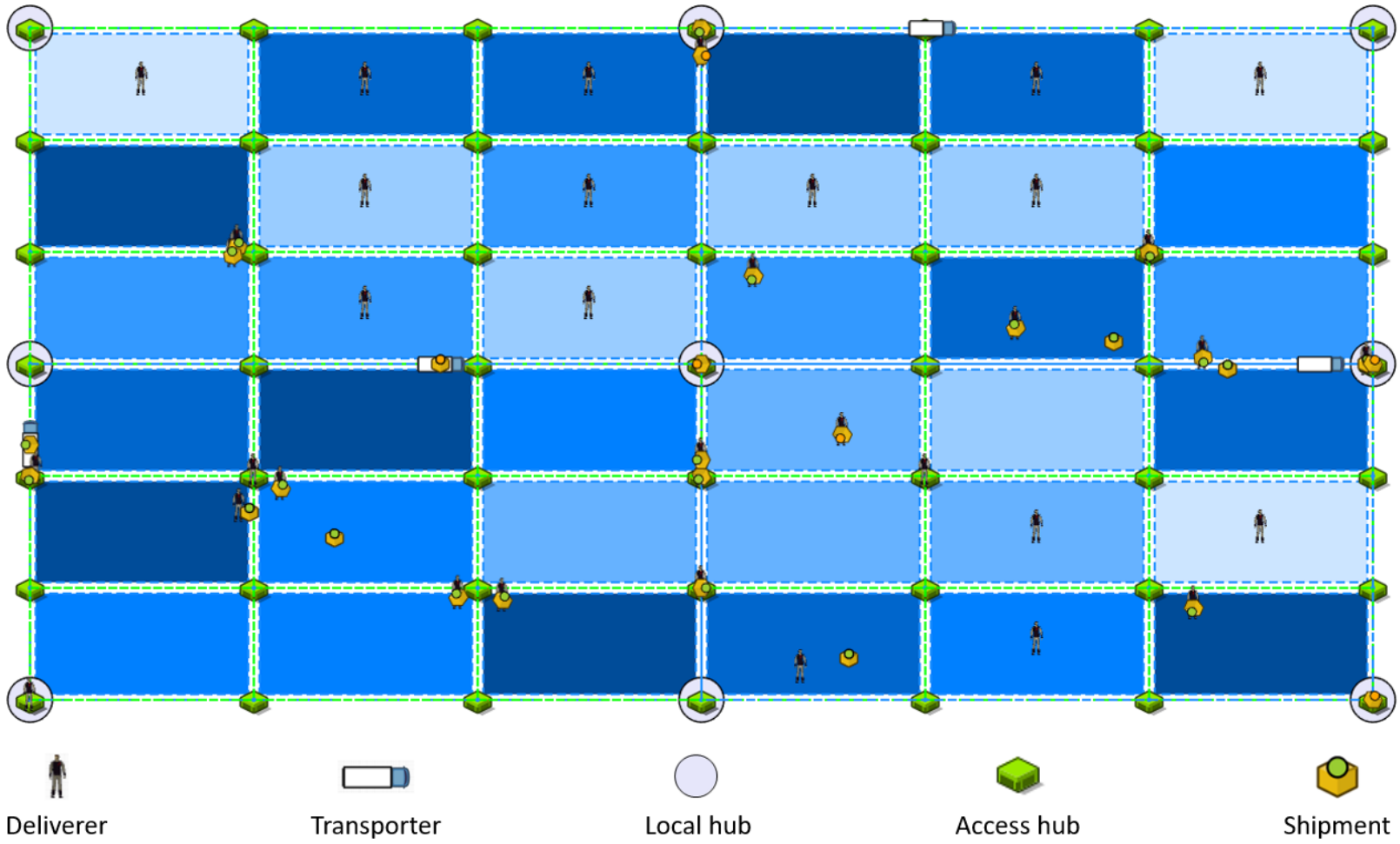


FIGURE 4.1 : Représentation d'une mégalopole et de ses composants (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

villes potentielles. Les *shuttlers* utilisent préférablement des routes plus rapides que celles empruntées par les *riders*, de la même façon que les *riders* prennent des routes plus rapides que les *couriers*. Dans un but de simplification, ici les *riders* et *shuttlers* sont rassemblés dans un groupe de *transporters* (transporteurs).

4.2.2 RÉSEAU ETHEREUM ET SMART CONTRACTS

Nous détaillons ici les choix de conception relatifs au réseau Ethereum qui sera mis en place en parallèle de la simulation AnyLogic de la Section 4.2.1. Celle-ci ne détaillant pas le traitement des colis à l'intérieur même des *hubs*, les seules actions qui seront prises en compte seront les ramassages (*pick-ups*) et livraisons (*deliveries*) effectués par les *deliverers* et les *transporters*. Ces deux entités constituent les *agents* de notre simulation.

UN RÉSEAU BLOCKCHAIN PRIVÉ

Dans cette simulation, le réseau blockchain est privé : cela signifie que plusieurs nœuds Ethereum sont lancés et communiquent entre eux, et que leur blockchain est indépendante des autres réseaux comme le *main net* [24] d'Ethereum (où de vraies cryptomonnaies sont utilisées) ou les *test nets* tels que Kovan [25] ou Rinkeby [26] (qui sont principalement utilisés par les développeurs pour tester leurs applications). Cela nous permet de personnaliser la configuration de la blockchain à notre guise. Par exemple, notre réseau utilise le consensus Proof-of-Authority pour construire les blocs de transactions (voir Section 5), et aucun des nœuds validateurs n'impose de restriction sur le prix de *gas* minimum requis. Ceci nous autorise

24. https://etherscan.io/
25. https://kovan.etherscan.io/
26. https://rinkeby.etherscan.io/

également à exécuter tous les nœuds localement, c.-à-d.sur le même ordinateur, éliminant ainsi les délais de soumission de transactions et d'émission de blocs liés à l'acheminement de paquets IP via Internet. De plus, ceci simplifie l'allocation de fonds aux comptes Ethereum des agents de la simulation.

Pour simuler un réseau tel qu'il serait mis en place dans la réalité, nous avons choisi d'attribuer un client Ethereum à chaque agent de la simulation AnyLogic. Cela signifie que chaque *deliverer* et *transporter* reçoit la totalité des transactions du réseau, les valide lors de la réception d'un bloc, les stocke, et donc garde en mémoire la blockchain en entier. De plus, nous exécutons en parallèle des *sealers*, c.-à-d. les nœuds validateurs d'un réseau utilisant PoA en consensus. Les *sealers* ne sont associés à aucun agent, mais peuvent plutôt être vus comme des serveurs mis en service par les diverses entreprises, organisations ou institutions pour assurer le bon fonctionnement du réseau. Comme nous l'avons mentionné dans la partie concernée, PoA implique que la majorité des *sealers* doivent être considérés fiables, ce qui importe peu étant donné qu'ils sont exécutés sur la même machine dans notre cas. Le choix de ce consensus est surtout motivé par le fait qu'il est bien plus léger que Proof-of-Work en termes de besoins calculatoires (un critère important puisque nous exécutons tous les nœuds sur la même machine).

SPÉCIFICATION DES SMART CONTRACTS

La plateforme Ethereum permet le déploiement de *smart contracts* compilés préalablement spécifiés en langage Solidity. Dans cette section nous allons justement définir les deux contrats utilisés pour stocker les actions réalisées sur les colis : les contrats `ShipmentManager` et `Shipment`. En l'occurrence, nous définissons une action comme une structure de données composée des éléments suivants :

— l'identifiant du colis manipulé ;

— la date et l'heure à laquelle l'action a eu lieu ;

— son type (dans ce scénario, nous ne prenons en compte que les ramassages et livraisons de colis) ;

— le nom et l'adresse Ethereum de l'agent effectuant l'action ;

— la localisation de l'action (c.-à-d., les coordonnées X-Y auxquelles a eu lieu l'action dans la simulation).

Contrat « Shipment » Un contrat `Shipment` représente un unique colis dans la simulation et stocke chaque action effectuée sur celui-ci. Il contient donc les éléments suivants :

— `id` : l'identifiant unique du colis (c.-à-d., son nom dans la simulation AnyLogic) ;

— `destination` : les coordonnées X-Y de la destination finale du colis ;

— `actions` : la liste des actions réalisées sur le colis ;

— `shipmentManager` : une référence au `ShipmentManager` du colis, dont le principe est expliqué plus tard.

En termes d'interface, un `Shipment` fournit également les trois méthodes suivantes :

— `addAction(action)` : ajoute l'action spécifiée à la liste des actions (`actions`). Seul le `ShipmentManager` du colis peut appeler cette méthode ;

— `getActionCount()` : retourne le nombre d'actions effectuées sur le colis jusqu'à présent ;

— `getAction(index)` : retourne une action en spécifiant son index dans la liste d'actions.

Contrat « ShipmentManager » Un `ShipmentManager` fournit une interface responsable de la gestion des contrats `Shipment` caractérisant les colis d'une même application ou contexte. Il est composé d'un seul attribut et de trois méthodes :

— `shipmentsById` : une structure recensant les contrats `Shipment` par l'identifiant du colis correspondant dans la simulation. Par conséquent, elle maintient une liste des références à tous les contrats `Shipment` du contexte ;

— `createShipment(shipmentId, destination)` : crée un nouveau contrat `Shipment` en fournissant son `shipmentId` et sa `destination` finale, et l'ajoute à la *map* `shipmentsById`. Nous supposons que cette méthode est appelée dès que l'envoi de colis est demandé dans la simulation. Chaque création produit un *log event* décrivant le nouveau colis concerné ;

— `addAction(shipmentId, action)` : appelle la méthode `addAction` du contrat `Shipment` référencé dans `shipmentsById` et qui correspond au `shipmentId` fourni, c.-à-d., l'identifiant du colis. De plus, un *log event* décrivant la nouvelle action est émis ;

— `getShipment(shipmentId)` : retourne l'adresse du contrat `Shipment` référencé dans `shipmentsById` et correspondant au `shipmentId` fourni, ou l'adresse nulle (c.-à-d., `0x0`) si le contrat n'existe pas encore.

On pourrait soutenir que le contrat `ShipmentManager` n'est pas nécessaire, et que les actions pourraient être directement envoyées aux contrats `Shipment`. Cependant, il y a plusieurs justifications pour ce choix.

Premièrement, utiliser seulement les contrats `Shipment` implique que chaque agent ait connaissance des adresses des contrats correspondants aux différents colis qu'il manipule, ce qui signifie qu'une base de données ou un fichier externe contenant ces adresses doit être maintenu et mis à disposition de tous les agents, et que ce système soit mis à jour à

chaque nouveau déploiement de `Shipment`. Avec un contrat `ShipmentManager`, ceci n'est plus nécessaire puisque les changements ou appels sont effectués à travers celui-ci, mettant automatiquement à jour les références des contrats `Shipment` : il n'y a donc pas besoin d'un système externe et l'adresse du `ShipmentManager` est tout ce que les agents doivent connaître pour ajouter des actions.

Deuxièmement, une telle interface permet d'émettre des événements lors de la création de nouveaux colis et l'ajout de nouvelles actions à partir d'un unique contrat `Shipment-Manager`. Autrement, chaque contrat `Shipment` serait responsable d'émettre ses propres événements. Si l'on souhaitait suivre les événements émis par un ensemble de colis, il faudrait le faire pour chaque `Shipment` souhaité. Dans notre solution, il faudrait simplement écouter les événements émis par le `ShipmentManager` et filtrer ceux considérés comme non pertinents.

Dernièrement, un contrat `ShipmentManager` permet d'effectuer des vérifications par rapport à un ensemble de colis. En effet, le traitement de certains d'entre-eux pourrait être contraint par des règles spécifiques (*p. ex.*, des actions sur certains colis pourraient être interdites ou autorisées seulement pour des agents spécifiques). Par conséquent, les mécanismes de contrôle d'accès, le respect de la conformité des données, ou tout autre type de vérification pourrait être implémenté directement au niveau du `ShipmentManager`, permettant ainsi d'établir différents niveaux de granularité pour les contrôles.

4.2.3 INTERACTIONS ENTRE LA BLOCKCHAIN ET ANYLOGIC

Jusqu'à présent, nous avons mis en place notre réseau blockchain et spécifié les contrats nécessaires. À l'initialisation du réseau, un `ShipmentManager` est déployé pour référencer tous les futurs contrats `Shipment` créés par la simulation. Cependant, AnyLogic ne fournit aucune fonctionnalité pour communiquer avec des clients Ethereum ou des *smart contracts*.

Des outils externes sont donc nécessaires pour que les agents à l'intérieur de la simulation puissent envoyer des actions vers la blockchain.

LA BIBLIOTHÈQUE WEB3J

Notre première démarche fut d'utiliser web3j [27], une bibliothèque Java open source permettant de communiquer avec des clients Ethereum via IPC directe (*Inter-Process Communication*, généralement utilisée lorsque l'application Java et le client Ethereum sont exécutés sur la même machine), ou par RPC (*Remote Procedure Call*, en utilisant le protocole HTTP). L'activation de ces options de communication doit être spécifiée au démarrage du client Ethereum pour que celles-ci soient autorisées. Avec web3j, un programme Java peut donc facilement se connecter à n'importe quel client Ethereum disponible et réaliser des transactions communes, telles que le transfert d'Ether vers un autre compte, le déploiement d'un *smart contract* ou l'appel d'une de ses méthodes.

L'un des principaux intérêts de web3j est qu'elle traite directement avec des *smart contracts* Java générés depuis leur code Solidity, facilitant grandement le déploiement de ceux-ci et leur manipulation. Reprenons l'exemple du contrat `MyStringStorageContract` présenté dans la Figure 3.6. Un exemple de manipulation d'un tel *smart contract* avec web3j et son résultat sont donnés dans la Figure 4.2 et la Figure 4.3, respectivement.

La classe Java `MyStringStorageContract` est issue d'un fichier généré à l'aide des outils [28] fournis avec web3j. Pour cela, le *smart contract* doit être d'abord compilé, directement via la plateforme Remix ou le compilateur Solidity solc, afin d'obtenir les fichiers `.bin`,

27. https://web3j.io/

28. https://docs.web3j.io/smart_contracts/#solidity-smart-contract-wrappers

```java
// Chaque unite de gas vaut 10 Wei (1 Wei = 10^-18 ETH)
public static final BigInteger GAS_PRICE = BigInteger.valueOf(10);

// Le cout de la transaction ne peut exceder 6,000,000 gas
public static final BigInteger GAS_LIMIT = BigInteger.valueOf(6000000L);


public static void main(String[] args) throws Exception {

    // Ininitialisaton de la connexion au noeud Ethereum
    Web3j web3j = Web3j.build(new HttpService("http://localhost:8545"));

    // Chargement du porte-monnaie Ethereum
    Credentials wallet = WalletUtils.loadCredentials("", "wallet.json");

    // Deploiement du smart contract
    MyStringStorageContract contract = MyStringStorageContract
            .deploy(web3j, wallet, GAS_PRICE, GAS_LIMIT, "Hello, you!")
            .send();

    // Affichage de l'adresse du contrat
    // Peut etre utilisee pour recuperer le contrat ulterieurement
    System.out.println(contract.getContractAddress());

    // Affichage de l'actuelle string du contrat
    System.out.println(contract.getMyString().send());

    // Modification de la string du contrat et affichage
    contract.setMyString("Hello, world!").send();
    System.out.println(contract.getMyString().send());
}
```

FIGURE 4.2 : Code Java pour manipuler un contrat MyStringStorageContract avec web3j (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

```
1 0x4985f73ecaacd710a087388389d54522ad4af868
2 Hello, you!
3 Hello, world!
```

FIGURE 4.3 : Sortie console du programme correspondant à la Figure 4.2 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

contenant le *bytecode* du contrat, et `.abi` (pour *Application Binary Interface*), décrivant l'interface de celui-ci. Ces deux fichiers sont ensuite utilisés pour produire une classe Java *encapsulant* le déploiement et la récupération du contrat, ainsi que son interface (c.-à-d., les méthodes du contrat pouvant être appelées).

Dans notre exemple de la Figure 4.2, le programme Java se connecte tout d'abord au client Ethereum disponible à l'adresse `http://localhost:8545/` via RPC (ligne 11). Puis, les identifiants (c.-à-d., l'adresse et la paire de clés publique/privée de l'utilisateur) sont récupérés depuis un fichier *wallet* (porte-monnaie Ethereum) en utilisant un mot de passe (vide dans ce cas) pour déchiffrer la clé privée (ligne 14). Un contrat `MyStringStorageContract` est ensuite déployé avec comme chaîne initiale "`Hello, you!`", ainsi qu'un prix et une limite de *gas* (voir Section 3.5.2) prédéfinis [29] (lignes 17–19). Une fois la transaction réussie, on peut consulter son adresse (ligne 23), qui est affichée en console (Figure 4.3, ligne 1). On peut également récupérer la valeur de la chaîne actuellement stockée dans le contrat (ligne 26) et l'afficher à son tour dans la console (Figure 4.3, ligne 2). Finalement, la chaîne est modifiée (ligne 28), et l'on peut vérifier sa valeur (ligne 30) en l'affichant (Figure 4.3, ligne 3).

29. En l'occurrence, le prix et la limite de *gas* définis ici vaudront également pour les transactions suivantes du programme. Il est néanmoins possible de définir des valeurs dynamiques s'adaptant au type de la transaction effectuée via un objet `ContractGasProvider`.

ARCHITECTURE DE LA SOLUTION

Puisque les simulations AnyLogic sont développées en Java, notre première idée fut d'importer web3j dans AnyLogic et gérer les interactions avec les *smart contracts* directement à l'intérieur de la simulation. Cependant, il y a un conflit entre la bibliothèque cryptographique déjà présente dans AnyLogic et celle requise par web3j, rendant cette option impossible. Il a donc fallu développer un programme en dehors d'AnyLogic qui serait capable de transmettre les actions émises par les agents à l'intérieur de la simulation vers les clients Ethereum de ces derniers (on rappelle qu'un client Ethereum est attribué à chaque agent).

Le programme intermédiaire responsable de la transmission des actions est appelé *Action Manager* : il s'agit en réalité d'un serveur HTTP auquel l'agent a fourni en amont son porte-feuille Ethereum (qui contient, entre autres, ses clés publique et privée) et le mot-de-passe correspondant. Le serveur attend les requêtes POST émises par l'agent contenant les informations des actions réalisées, puis les traite et les envoie via RPC au client Ethereum en utilisant web3j. Comme illustré dans la Figure 4.4, trois types d'entités sont donc impliqués : 1) l'agent à l'intérieur de la simulation AnyLogic qui envoie le détail des actions ; 2) l'*Action Manager* qui reçoit ces actions et les transmet dans des transactions en appelant la méthode `addAction` du `ShipmentManager` déployé auparavant ; 3) le nœud Ethereum qui reçoit ces transactions et les communique au reste du réseau Ethereum pour qu'elles soient intégrées dans un bloc.

PASSAGE AU LANGAGE GO

La solution mentionnée plus tôt, bien que fonctionnelle, avait le défaut de consommer trop de mémoire (rappelons que la simulation ainsi que la totalité des *Action Managers* et des

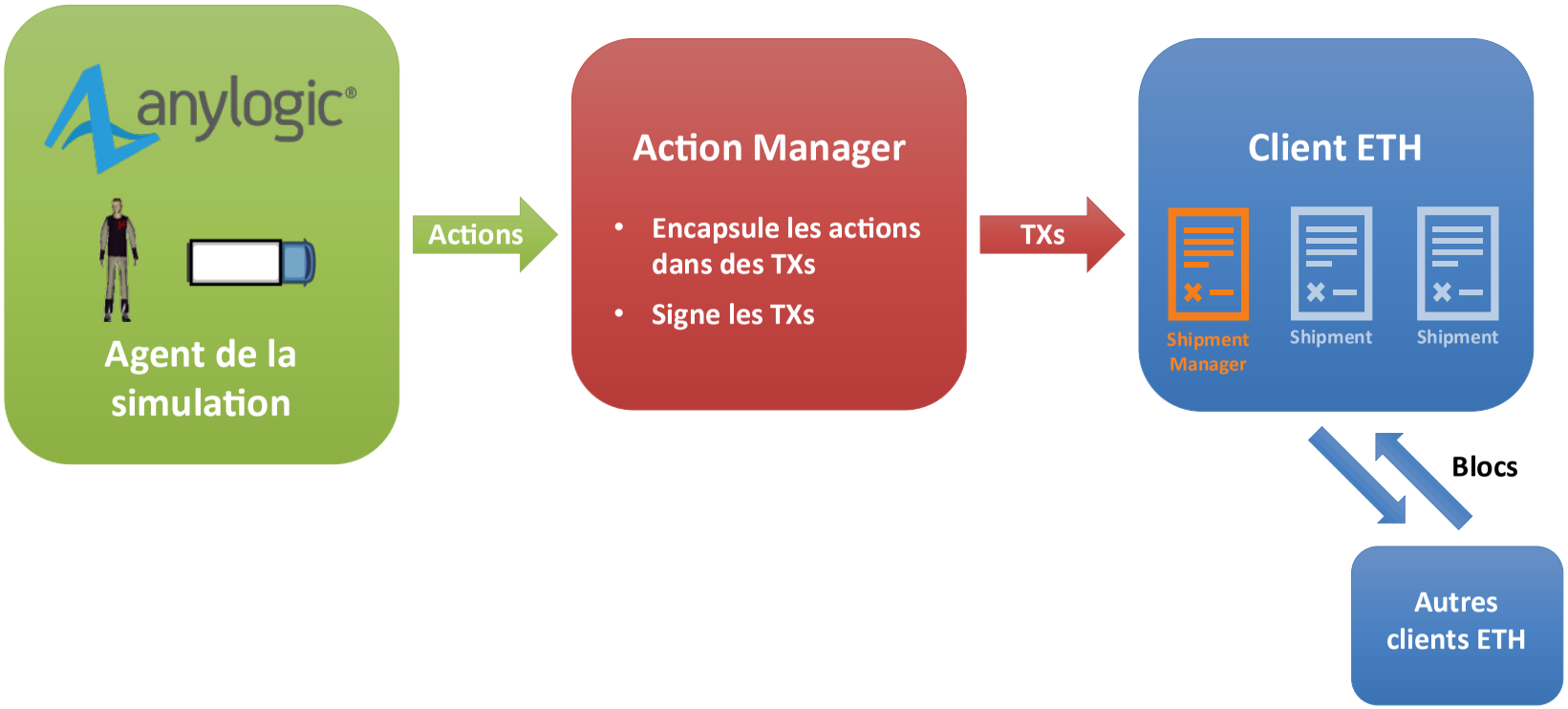


FIGURE 4.4 : Architecture de la solution pour un agent (TXs = transactions, ETH = Ethereum) (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

clients Ethereum sont exécutées sur la même machine). En effet, un unique *Action Manager* consommait jusqu'à 200 Mo de mémoire, et 40 d'entre-eux tournent en simultané (il y a 40 agents – 36 *deliverers* et 4 *transporters* – dans la simulation). Sur un ordinateur standard, allouer près de 8 Go de mémoire uniquement pour les *Action Managers* peut provoquer plusieurs exceptions de type `OutOfMemoryException`, occasionnant l'arrêt soudain de ceux-ci.

Afin de réduire leur consommation en mémoire, il fut décidé de passer de programmes Java à des programmes en langage Go, qui est le langage natif des clients Ethereum Geth et permet une gestion très légère des requêtes HTTP. De plus, la bibliothèque incluse dans Geth, go-ethereum, est également open source et offre les mêmes fonctionnalités que web3j. Les *Action Managers* furent donc re-codés en Go en utilisant go-ethereum. Ce changement eut des résultats largement positifs, puisque l'exécution des *Action Managers* requiert maintenant seulement 20 Mo de mémoire, contre 8 Go pour l'implémentation précédente.

Notons d'ailleurs que l'architecture globale choisie permet d'effectuer ce type de transitions très facilement. En effet, si un retour aux *Action Managers* Java était nécessaire, il suffirait de remplacer les *Action Managers* écrits en Go par ceux-là, sans qu'aucun changement dans la simulation en elle-même ne soit nécessaire. Les *Action Managers* pourraient d'ailleurs faire l'objet d'autres modifications sans que cela n'impacte le bon fonctionnement de la simulation, à l'unique condition que l'implémentation choisie permette le traitement de requêtes HTTP et l'interaction avec des clients Ethereum.

4.3 PROPRIÉTÉS DE LOGISTIQUE HYPERCONNECTÉE

Comme nous l'avons mentionné dans la Section 1.2, le paradigme des logistiques hyperconnectées offre de nombreux avantages. Cependant, il possède également des inconvénients

qu'il est important de noter. Le fait que bien plus d'acteurs soient impliqués dans le transport d'un unique colis augmente les risques d'erreurs ou d'actions malveillantes délibérées. En effet, un colis peut être plus facilement « caché » ou « perdu » puisqu'il change de mains plus fréquemment et que chaque colis a son propre cycle de vie. Dans ce cas, par cycle de vie, nous désignons le traitement « correct » d'un colis qui est, en fait, le mieux caractérisé par les étapes qu'il devrait suivre et les contraintes devant être respectées tout au long de son transport. En outre, il est plus difficile de savoir quel acteur est chargé du colis à un moment précis, ce qui rend le suivi effectif des colis encore plus ardu. Dans les faits, introduire une forme de décentralisation dans la gestion de la *supply chain* complexifie l'obtention d'*instantanés* fidèles de son état global. Il est donc nécessaire de spécifier plusieurs « contraintes » devant être respectées en tout temps par un unique colis ou pour la totalité du réseau, et qui sont indépendantes de la topologie de ce dernier ou de la nature des paquets acheminés.

D'autre part, avec le système basé sur une blockchain mis en place dans la Section 4.2, il devient possible d'externaliser le fait de conserver dans la blockchain un registre d'actions global et cohérent, puis de lire les éléments de celle-ci lorsqu'ils sont ajoutés, en se basant sur les événements émis par les *smart contracts*. Ce registre forme l'*historique* de notre *supply chain* et des colis en jeu. Celui-ci est donc assimilable à un *flux* composé d'événements caractérisant les étapes parcourues par l'ensemble des colis en tout temps.

Dans cette section, nous exprimons les propriétés qu'il nous semble pertinent de vérifier dans le cadre des logistiques hyperconnectées. Les premières sont des propriétés d'*exactitude* (*correctness properties*), dont la violation implique généralement un dysfonctionnement ou une intervention malveillante dans la *supply chain*. Puis, nous introduisons deux types de propriétés moins étudiés dans la littérature. Premièrement, les propriétés *analytiques* sont non-booléennes (c.-à-d., leur vérification ne résulte ni en un « vrai », ni en un « faux »), ce ne sont donc pas des propriétés à proprement parler ; elles fournissent des informations

exploitables quant au statut de la chaîne, telles que la distribution statistique du temps de livraisons entre deux points. Deuxièmement, nous présentons les propriétés *auto-corrélées*, c.-à-d. des requêtes paramétrées qui comparent une tendance calculée sur une partie récente du journal d'événements à son propre historique.

4.3.1 PROPRIÉTÉS D'EXACTITUDE

Dans un système où les statuts de multiples paquets sont observés en même temps, il est souhaitable de s'assurer que les événements relatifs à une unique instance de colis suivent une séquence appropriée d'événements *pick-up* (ramassage) et *delivery* (livraison). Pour rappel, ces événements sont stockés dans la blockchain par les agents à l'aide des *smart contracts* `ShipmentManager` et `Shipment` (définis en Section 4.2.2) ; un *pick-up* correspond au ramassage d'un colis dans *hub* par un agent, tandis qu'un *delivery* indique que l'agent a déposé le colis dans un *hub* intermédiaire ou l'a livré à sa destination finale. Ceci donne lieu à la première propriété d'exactitude :

Propriété 1. *Un colis doit suivre un cycle de vie précis, où : 1) un événement pick-up doit nécessairement être suivi d'un événement delivery, et vice versa ; 2) un événement pick-up doit avoir lieu au même emplacement que le delivery précédent (c.-à-d., un colis ne devrait pas avoir bougé entre temps) ; 3) le dernier événement est un delivery à la destination finale attendue.*

Les potentielles déviations de ce cycle de vie pourraient avoir plusieurs causes. Par exemple, un employé pourrait simplement avoir oublié de scanner un colis lors de son ramassage ou d'une livraison. D'autres circonstances, plus néfastes, telles que l'altération malveillante de l'identifiant d'un colis ou le déplacement de l'un d'eux sans être scanné, pourraient également être détectées à travers la violation de ce cycle.

Un paquet peut également être ré-aiguillé pendant son transit. Cela signifie qu'une nouvelle destination est déclarée pour celui-là ; un tel ré-aiguillage peut être détecté en observant les ensembles de coordonnées X-Y des différentes destinations du colis entre deux événements successifs. Cependant, le nombre de ré-aiguillages que peut subir un colis pourrait être limité par une constante prédéfinie. Ceci mène à la contrainte suivante :

Propriété 2. *La destination d'un colis (c.-à-d., les coordonnées X-Y de la destination finale) ne doivent pas changer plus de k fois avant que celui-ci n'atteigne sa destination.*

De plus, on pourrait également empêcher que la destination du colis résulte en un va-et-vient entre plusieurs destinations, menant à une autre contrainte :

Propriété 3. *Chaque ré-aiguillage doit envoyer le colis vers une nouvelle destination.*

Puisque le but des *supply chains* connectées est d'acheminer les colis à leur destination de façon efficace, une autre propriété devrait stipuler que ceux-ci ne doivent pas emprunter de détours sur leur chemin :

Propriété 4. *Un colis doit toujours progresser vers sa destination finale.*

Superviser une telle propriété peut s'avérer utile pour détecter des colis prenant un chemin suspect qui les éloigne de leur destination attendue.

Il peut être également intéressant de limiter la vitesse de déplacement d'un colis entre deux étapes. Il existe effectivement une limite maximale raisonnable, due aux temps de traitement et vitesses des différents moyens de transport impliqués, sur la distance qu'un colis peut parcourir en un certain temps :

Propriété 5. *La vitesse d'un colis ne peut excéder une certaine valeur maximale v sur un intervalle de temps.*

La violation d'une telle propriété pourrait indiquer que la position du colis a été altérée de manière illégitime, ou qu'il y a une défaillance dans le système de mesure de position. Le raisonnement inverse peut être également réalisé sur la vitesse minimale d'un colis :

Propriété 6. *Les événements relatifs à un colis ne devraient pas être séparés par plus de x unités de temps.*

Cette propriété implique la notion de *délai* (*timeout*) ; elle est utilisée pour émettre un avertissement lorsqu'aucune nouvelle concernant un colis n'a été reçue après un temps prédéfini. Ceci peut notamment servir à détecter les colis perdus.

4.3.2 PROPRIÉTÉS COMPLEXES

Les propriétés que nous avons mentionnées jusqu'à présent sont relativement représentatives des types de contraintes habituellement étudiées dans la littérature concernant le runtime verification et produisent des verdicts booléens, c.-à-d. la propriété est vérifiée ou non. Cependant, le scénario de logistique hyperconnectée que nous avons introduit plus tôt ouvre la voie à d'autres types de calculs sur les événements relatifs aux colis qui ne rentrent pas dans le cadre de cette définition. Nous nous tournons donc vers des propriétés plus « avancées » pouvant être également monitorées.

PROPRIÉTÉS ANALYTIQUES

Premièrement, nous qualifions d'« analytique » toute propriété dont le résultat n'est pas un verdict booléen. Ces propriétés peuvent être utilisées, par exemple, pour calculer certaines statistiques sur l'état de la *supply chain*. Par exemple :

Propriété 7. *Calculer le nombre de colis qui sont actuellement en transit, c.-à-d. qui n'ont pas encore atteint leur destination.*

Propriété 8. *Calculer la distribution du temps requis pour acheminer un colis pour une paire source–destination donnée.*

Propriété 9. *Calculer le nombre de colis qui transitent via une zone donnée.*

Ces propriétés peuvent être d'une aide précieuse pour évaluer la qualité et l'efficacité du réseau de logistique, et donc pour décider de solutions et d'améliorations appropriées. Par exemple, si le résultat de la Propriété 9 est beaucoup plus élevé que la moyenne pour une zone, cela pourrait indiquer qu'il faudrait y attribuer plus de livreurs/*hubs*, ou la subdiviser en plusieurs zones pour assurer une efficacité optimale du réseau.

PROPRIÉTÉS AUTO-CORRÉLÉES

Toutes les propriétés d'exactitude vues jusqu'ici sont « absolues », dans le sens que la même condition doit s'appliquer à tous les événements, ou chaque sous-flux d'événements pour chaque colis. Par exemple, la Propriété 2 impose un nombre maximum de fois où la destination d'un paquet peut être changée ; ce nombre est connu à l'avance et s'applique à tous les colis. Il existe d'autres propriétés où ce qui est considéré comme « correct » dépend en réalité d'agrégations calculées sur des fenêtres d'événements passés d'un même flux ; dans un tel cas, une référence statique ne peut être établie à l'avance. Nous qualifions ces propriétés d'« auto-corrélées », en ce sens qu'au moins une partie de leurs paramètres déterminants est basée sur le flux lui-même.

Un bon exemple de ce type de propriété concerne le temps de livraison de colis sur un seul intervalle (*p. ex.*, transport entre deux *hubs*). Il est difficile de déterminer à l'avance

ce qui constitue un temps d'acheminement « normal » pour chaque intervalle du réseau hyperconnecté ; de plus, cette durée normale pourrait changer avec le temps, ce qui rendrait la définition d'une référence statique caduque. Cependant, il pourrait être intéressant d'observer les potentiels grands écarts avec ce qui était habituellement vu dans le passé. Cela peut être exprimé de la façon suivante :

Propriété 10. *Le temps moyen de livraison de m colis entre deux points donnés ne peut excéder d'un facteur k le temps moyen des n colis précédents entre ces deux mêmes points.*

Ce type de propriété, si monitoré en temps réel, pourrait notamment prévenir d'un ralentissement soudain dans l'acheminement des paquets. Les causes pourraient être variées ; par exemple, une voie d'accès majeure de l'intervalle pourrait être inaccessible (*p. ex.*, embouteillage dans une rue, fermeture exceptionnelle d'un aéroport due à un incident), ou un dysfonctionnement technique pourrait avoir eu lieu dans une plateforme de tri, retardant la prise en charge des colis. Pouvoir identifier rapidement et avec exactitude les incidents au sein d'un réseau logistique est une étape essentielle pour proposer et trouver des solutions optimales, comme le ré-aiguillage d'un ensemble de colis vers une autre zone pour éviter le ralentissement.

4.4 SUPERVISION DES PROPRIÉTÉS AVEC BEEPBEEP

Maintenant que nous avons spécifié des propriétés dont la vérification et supervision sont pertinentes dans le cadre de la logistique hyperconnectée, nous pouvons utiliser les événements émis par les *smart contracts* définis dans la Section 4.2.2 comme trace source pour les vérifier. Pour ce faire, nous allons implémenter ces propriétés en utilisant BeepBeep, la bibliothèque de *Complex Event Processing* présentée en Section 2.2.1. Cependant, il est

tout d'abord nécessaire de montrer comment BeepBeep est capable d'interagir avec le réseau Ethereum pour en extraire les actions permettant la vérification de ces propriétés.

4.4.1 INTERACTIONS ENTRE BEEPBEEP ET ETHEREUM

Afin de permettre la vérification des propriétés présentées précédemment, il faut fournir à BeepBeep un moyen d'accès aux informations contenues dans la blockchain Ethereum, ce qui n'est pas permis par ses fonctionnalités de base.

Comme nous l'avons montré plus tôt, web3j permet, entre autres, de déployer de nouveaux *smart contracts* sur une blockchain, de leur envoyer des transactions, et d'écouter les *log events* qu'ils émettent. Ce faisant, il est possible d'utiliser web3j pour « attraper » tout événement émis par un contrat spécifique et l'utiliser pour un traitement ultérieur.

BeepBeep et web3j étant tous deux écrits en Java, nous avons donc implémenté une palette Ethereum pour BeepBeep [30]. En fait, celle-ci encapsule certaines fonctionnalités fournies dans web3j pour qu'elles puissent être utilisées dans les chaînes de processeurs réalisées avec BeepBeep. Plus particulièrement, elle permet d'utiliser les *log events* émis par des *smart contracts* comme source d'événements en entrée des chaînes. Le code suivant montre comment un processeur BeepBeep spécial, appelé `CatchEthContractLogs`, peut être connecté à une blockchain et écouter des événements émis par un contrat spécifique :

```
CatchEthContractLogs source = new CatchEthContractLogs(
  "http://localhost:8545", "0x6702413C52c8Cf0fc5f", true);
```

Le premier paramètre est l'URL du nœud Ethereum auquel on souhaite se connecter via RPC ; dans notre exemple, le nœud est exécuté localement et est accessible via le port

30. https://github.com/liflab/bb-palette-blockchain

TCP `8545`. Le second paramètre est l'adresse Ethereum du *smart contract* à écouter. Enfin, le paramètre booléen `true` indique que le processeur captera tous les événements émis par le contrat, depuis le premier le bloc de la blockchain. Le mettre à `false` résulterait en un processeur qui ne reporterait que les *nouveaux* éléments ajoutés à la blockchain, ultérieurs à l'instanciation du processeur.

Ce processeur agit tel un processeur `Source` de BeepBeep et, de fait, hérite de cette classe. Ainsi, les instances de `CatchEthContractLogs` peuvent être utilisées comme n'importe quel autre processeur BeepBeep ; elles peuvent donc être connectées à d'autres processeurs qui effectueront des traitements ultérieurs sur les événements blockchain. Un `CatchEthContractLogs` est conçu de façon à ce que, lors de l'appel de sa méthode `start` [31], tous les entrées correspondantes dans la blockchain seront « poussées » en aval d'un seul coup (ou en goutte-à-goutte, si le troisième paramètre du constructeur est égal à `false`).

Puisqu'un *smart contract* peut émettre des événements variés, chacun ayant une structure propre, il serait pertinent de pouvoir filtrer ceux-ci afin de ne garder que ceux qui nous intéressent. C'est dans ce but que la palette définit également un objet BeepBeep `Function` appelé `GetEventParameters`. Cette fonction est instanciée avec un *objet filtrant* ; tout élément de la blockchain correspondant au nom et à la structure de l'objet sera gardé, tandis que les autres seront éliminés. Par exemple, supposons qu'un événement appelé `Student`, contenant le nom et l'âge d'un étudiant à travers, respectivement, les paramètres `name` et `age`, soit émis par un *smart contract* déployé sur la blockchain. Un tel événement serait déclaré de la façon suivante dans un *smart contract* Solidity :

31. Une méthode fournie par la classe de premier niveau `Processor`, qui peut être redéfinie pour réaliser différentes tâches, dépendant du rôle du processeur. Le comportement par défaut est de ne rien faire.

```
event Student(
  string name,
  uint age
);
```

En utilisant BeepBeep et sa palette Ethereum, il est alors possible de retenir seulement les événements de ce type avec le bloc de code suivant :

```
ApplyFunction filter = new ApplyFunction(new GetEventParameters(
  new Event("Student", Arrays.asList(
    new TypeReference<Utf8String>() {},
    new TypeReference<Uint256>() {} ))));
```

Ce code instancie un processeur `ApplyFunction`, dont la fonction est une instance de `GetEventParameters`. On fournit à celle-ci une instance d'objet `Event` (une classe fournie par la bibliothèque web3j) qui définit le nom et la structure interne du *log event* à chercher dans la blockchain.

À partir de là, il est possible relier la source définie plus tôt et ce filtre :

```
Connector.connect(source, filter);
Connector.connect(filter, new Print());
source.start();
```

Dans ce code, la source est connectée au filtre, et celui-ci est lui-même connecté à un processeur `Print` qui, simplement, écrit dans la console les événements qu'il reçoit en entrée. En démarrant le processeur `source` en appelant sa méthode `start`, toutes les entrées correspondantes ajoutées à la blockchain seront affichées dans la console en temps réel.

Nom	Description
Propriété 1	Un événement de type *pick-up* doit être suivi d'un événement de type *delivery* et vice-versa. Un *pick-up* doit avoir lieu au même endroit que le précédent *delivery*. Le dernier événement est un *delivery* à la destination attendue.
Propriété 2	La destination d'un colis devrait changer au plus k fois avant qu'il n'atteigne sa destination.
Propriété 3	Chaque ré-aiguillage doit envoyer le colis à une nouvelle destination.
Propriété 4	Un colis doit toujours progresser vers sa destination.
Propriété 5	La vitesse de déplacement d'un colis ne peut excéder une certaine limite sur un seul intervalle.
Propriété 6	Les événements sur un colis ne devraient pas être séparés par plus de x unités de temps.
Propriété 7	Calcule le nombre de colis actuellement en transit.
Propriété 8	Calcule la distribution du temps requis pour transporter un colis pour une paire source–destination donnée.
Propriété 9	Calcule le nombre de colis traversant une zone donnée.
Propriété 10	Le temps moyen de temps de livraison de m colis sur un seul intervalle ne doit pas excéder par un facteur k le temps moyen pour les n colis précédents sur le même intervalle.

TABLEAU 4.1 : Résumé des propriétés de cycle de vie définies dans la Section 4.3 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

4.4.2 IMPLÉMENTATION DES PROPRIÉTÉS

Il est maintenant possible d'implémenter les propriétés présentées plus tôt et résumées dans le Tableau 4.1 sous la forme de chaînes de processeurs BeepBeep. La plupart d'entre elles seront illustrées par les chaînes correspondantes. En accord avec la charte graphique de BeepBeep, des couleurs spécifiques seront associées aux événements selon leur type : les tuples (jaune), les chaîne de caractères (violet), les nombres (bleu-vert), les booléens (bleu-gris), les tableaux associatifs (bleu foncé) et les structures de données composées telles que les ensembles et les listes (rose à pois).

La Propriété 1 est adéquatement modélisée par le processeur BeepBeep `MooreMachine`, qui permet à l'utilisateur de définir des instances de machines de Moore, présentées en Section 2.1.1. La chaîne BeepBeep correspondante est montrée dans la Figure 4.5 et la machine de Moore en elle-même est représentée dans le contenu de la boîte #1 ; comme on peut le voir, les états de cette machine sont étiquetés avec les valeurs $\top$ (vrai), $\bot$ (faux) et « ? » (non concluant). Les transitions de cette machine sont associées aux fonctions à évaluer sur un événement entrant ; pour un état donné, une transition se déclenche si la fonction correspondante retourne $\top$ pour l'événement en entrée.

Une version étendue de la machine de Moore possède également des *variables d'état*, qui peuvent être modifiées lors d'une transition. Par exemple, la transition associée à la boîte #2, si empruntée, donnera à une variable interne appelée x la valeur de l'attribut *loc_x* dans l'événement courant (de même pour y). Ces variables d'état peuvent également être requises dans une transition, telle que la boîte #3, qui vérifie que les valeurs de *loc_x* et *loc_y* de l'événement courant sont bien égales aux précédentes valeurs sauvegardées dans les variables d'état x et y.

Puisque ce cycle de vie s'applique à chaque colis individuellement, un processeur `Slice` crée un sous-flux et instancie une machine pour chacun d'entre eux, en les distinguant sur leur *id* (boîte #4). La sortie du processeur `Slice` est un tableau associatif entre les *clés* des sous-flux (dans ce cas, les identifiants des colis) et le dernier événement renvoyé par chaque sous-processeur du `Slice` (dans ce cas, la valeur booléenne produite par la machine de Moore correspondante). Pour transformer cette chaîne en une propriété d'exactitude, la dernière étape est d'extraire les valeurs du tableau associatif et de réaliser une conjonction entre ces valeurs ; c'est la tâche effectuée par la boîte #5. Le résultat final est un flux de booléens qui retournera faux ($\bot$) exactement quand un des colis a violé son cycle de vie.

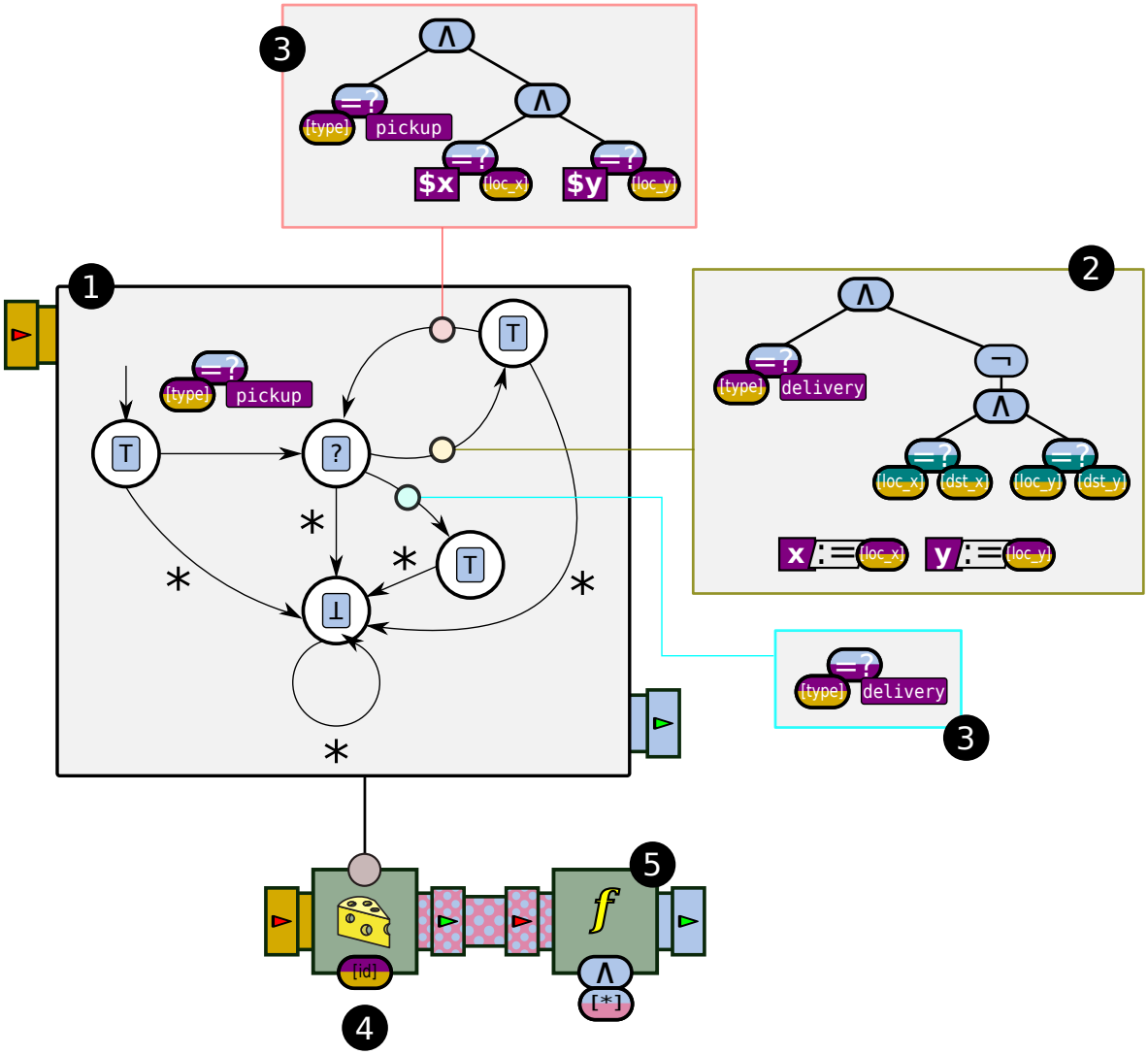


FIGURE 4.5 : La chaîne de processeurs pour la Propriété 1 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

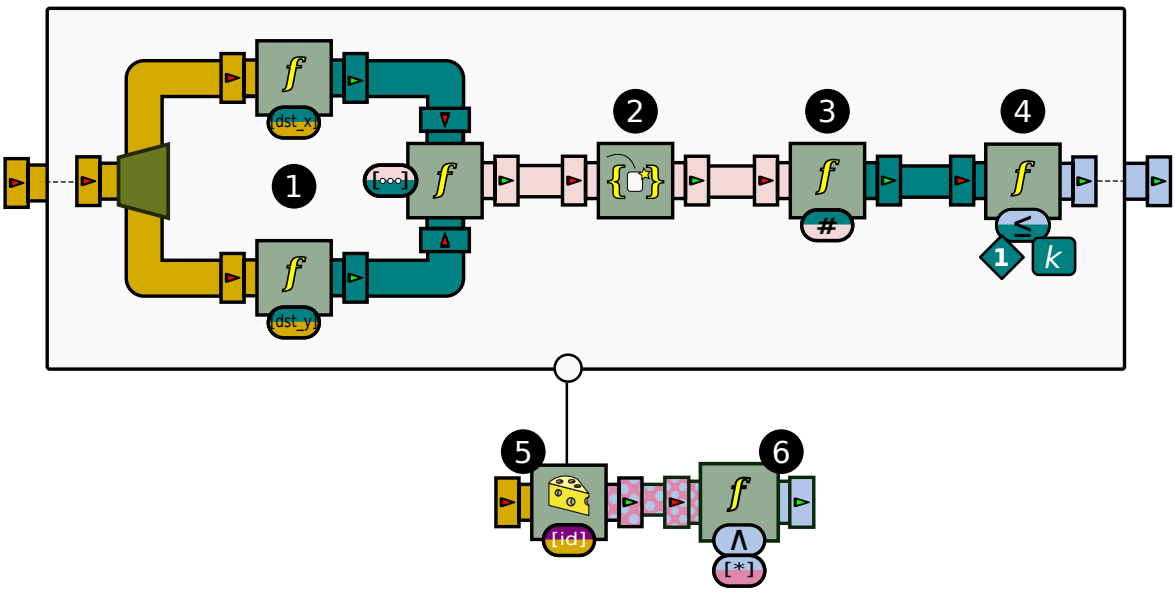


FIGURE 4.6 : La chaîne de processeurs pour la Propriété 2 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

La Propriété 2 peut être implémentée par la chaîne de processeurs de la Figure 4.6. À partir d'un événement donné, les attributs *destination_x* et *destination_y* sont extraits afin de créer un tuple *x*-*y* (#1 dans la figure). Ces tuples sont accumulés dans un ensemble (boîte #2), la cardinalité de ce dernier est récupérée (boîte #3) et comparée à une valeur *k* prédéfinie (boîte #4). Ce processus est répété pour tout identifiant de colis *id* via le processeur `Slice` (boîte #5), et la conjonction des valeurs du tableau associatif résultant est extraite (#6).

La Figure 4.7 montre la chaîne de processeurs associée à la vérification de la Propriété 3. La partie #1 de la chaîne extrait les coordonnées *x*-*y* de la destination d'un colis dans un événement ; la chaîne #3 au-dessus détecte lorsque deux événements successifs ont une destination différente. La chaîne #2 en bas accumule les tuples des destinations dans un ensemble, et vérifie si la destination courante était déjà présente l'ensemble précédent ; cela est réalisé par l'insertion d'une unique instance de $\emptyset$ dans le flux, afin de déphaser d'un événement

l'élément et l'ensemble à comparer. Les flux de booléens dans les deux branches sont ensuite combinés dans une implication (#4) ; le résultat est un flux retournant $\perp$ si et seulement si un changement de destination a eu lieu, et que la nouvelle destination est présente dans l'ensemble des destinations antérieures. Les processeurs #5 et #6 sépare respectivement le flux d'événements selon l'identifiant *id* des colis, et réalise la conjonction de tous les sous-flux, comme mous l'avons déjà vu plus tôt.

L'implémentation de la Propriété 4 est représentée par la chaîne de processeurs de la Figure 4.8. D'abord, tout événement qui n'est pas de type *delivery* est filtré du flux ; ceci est accompli par les processeurs #1 et #2. Le flux restant est séparé selon l'identifiant *id* de chaque colis (#5). Pour chacun d'eux, la distance entre sa position actuelle et sa destination est calculée (#3), et les distances entre deux événements successifs sont comparées (#4). Par conséquent, le processeur #5 produit un tableau associatif dans lequel les clés sont les *id* des colis, et les valeurs sont des booléens indiquant si le colis a progressé vers sa destination dans les deux derniers événements le concernant. Le processeur #6 réalise la conjonction de ces valeurs, faisant de cette chaîne une propriété d'exactitude.

La Propriété 7 peut être vérifiée en utilisant une version modifiée de la Figure 4.5. Au lieu de produire des booléens, la machine de Moore de la boîte #1 peut être adaptée pour envoyer le symbole 1 tant que le colis est en transit, et 0 quand il atteint sa destination. À la place de la conjonction, le bloc #5 peut calculer la somme de ces valeurs, ce qui correspondra donc au nombre de colis actuellement en transit.

Pour la Propriété 8, nous considérons une version simplifiée de celle-ci dans laquelle nous supposons qu'aucun colis n'est ré-aiguillé. En effet, dans ce cas la difficulté reposerait dans le fait que la paire source–destination serait vue uniquement dans le premier événement pour un colis donné ; les événements postérieurs y seraient certes liés par l'identifiant du

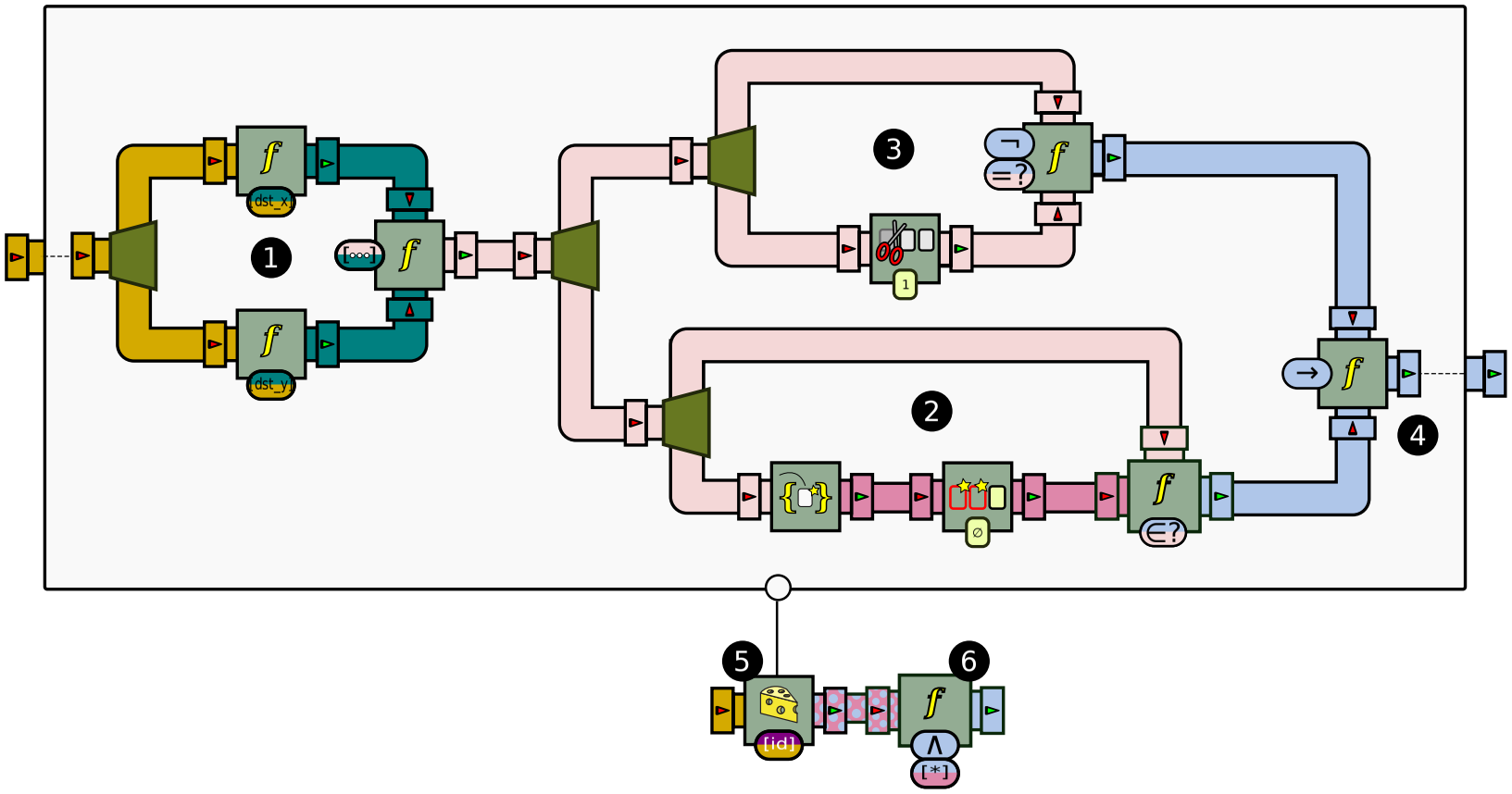


FIGURE 4.7 : La chaîne de processeurs pour la Propriété 3 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

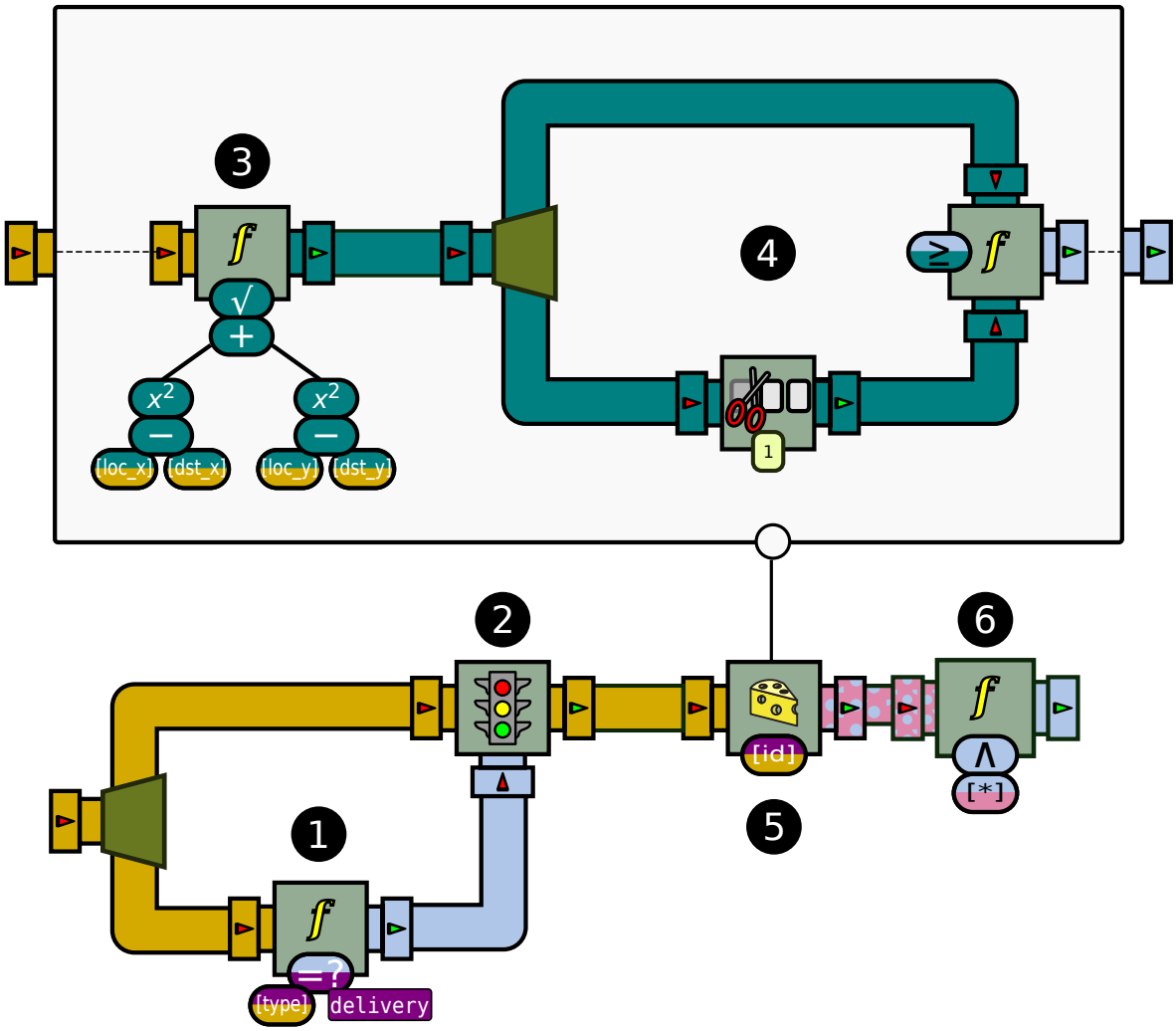


FIGURE 4.8 : La chaîne de processeurs pour la Propriété 4 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

colis, mais ceux-ci auraient une paire de coordonnées *x*-*y* différentes pour la même source. Ce cas mis à part, cette propriété peut être implémentée via la Figure 4.9. Une machine de Moore est instanciée pour chaque identifiant de colis (bloc #1) ; lors de sa création ; la machine sauvegarde dans ses variables d'état *x*, *y* et *t*, respectivement les coordonnées *x*-*y* de la destination du colis et l'horodatage de l'événement. Elle stocke également dans une variable *b* une valeur booléenne indiquant si les paires de coordonnées *x*-*y* source–destination correspondent à des valeurs fixes x_s-y_s et x_d-y_d (boîte #2). Si *b* est fausse, la machine transite vers un *état puits* (c.-à-d., un état non-acceptant depuis lequel toute transition renvoie vers lui-même) au prochain événement, et y reste sans produire de symbole. Si *b* est vraie, la machine « attend » dans l'état initial jusqu'à ce qu'un événement *delivery* pour lequel les coordonnées *x*-*y* correspondent à sa destination finale soit observé. Lorsque c'est le cas (boite #3), la variable d'état *d* stocke la différence de temps entre le premier horodatage et l'actuel, puis la machine transite vers un état produisant la valeur de *d* en symbole de sortie. Ensuite, la machine transite vers l'état puits et y reste.

Finalement, on a donc une Machine de Moore qui renvoie une durée de transport pour un colis si celui-ci a la paire de source–destination attendue, sinon rien. Le processeur `Slice` de la boîte #3 crée une instance de cette machine pour chaque identifiant distinct de colis ; le pictogramme en forme de drapeau indique une variante du `Slice` qui sort les événements directement produits dans chaque processeur interne, sans les accumuler dans un tableau associatif. Cela a pour résultat un flux de temps de transport, un pour chaque colis avec la paire source–destination désirée. Ces nombres sont accumulés dans un autre processeur `Slice` (bloc #4) ; cette fois, les événements sont séparés par intervalle de temps : une « tranche » est arbitrairement créée pour chaque valeur de $\lfloor \frac{x}{100} \rfloor$. En d'autres termes, la première tranche regroupera les temps entre 0 et 100, la deuxième les valeurs entre 100 et 200, et ainsi de suite. Le processeur qui s'exécute pour chaque tranche est un simple compteur (boîte #5). Ainsi,

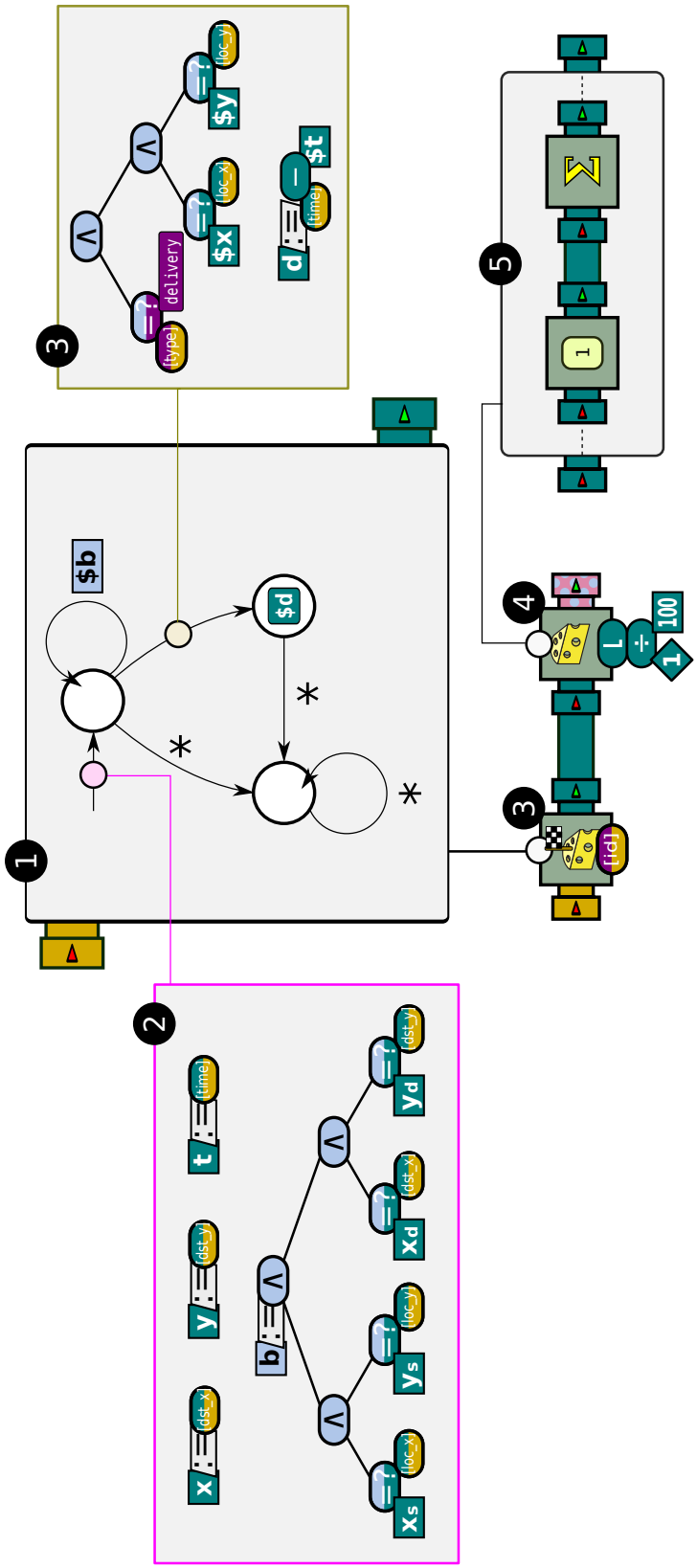


FIGURE 4.9 : La chaîne de processeurs pour la Propriété 8 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

le résultat de cette chaîne est un flux de tableaux associatifs ; dans chacun d'eux, une entrée $d \mapsto n$ indique que, jusqu'à présent, n colis dont le temps de transport appartient à la catégorie d ont été observés. Ce tableau peut ensuite être utilisé pour des traitements ultérieurs, par exemple en l'affichant sous la forme d'un histogramme [32]. On pourrait imaginer des variantes de cette propriété, telles que le calcul de la distribution du temps de transport pour tous les segments correspondant à un unique intervalle dans la grille.

La Propriété 10 peut être vérifiée en deux étapes. La première est de déterminer le temps de transport de chaque colis sur chaque intervalle unique. Cela est fait par la définition d'une machine de Moore, montrée dans la partie gauche de la Figure 4.10. À sa création, la machine stocke dans ses variables d'état *x*, *y* et *t* les coordonnées *x*-*y* et le l'horodatage de l'événement courant. Si la Propriété 1 est respectée, on peut supposer que le premier événement est un *pick-up*. Pour la même raison, le prochain événement doit donc être un *delivery* ; cette transition provoque le stockage par la machine des coordonnées de la destination du colis dans les variables d'états *x'* et *y'*, ainsi que la durée (différence entre l'horodatage de l'événement actuel et *t*) dans la variable *d*. Un symbole est alors émis, qui dans ce cas est un tuple. La valeur de l'attribut *hop* de ce tuple est elle-même une liste de quatre nombres, correspondant aux paires de coordonnées *x*-*y* de la source et de la destination de cet intervalle. La valeur de l'attribut *dur* contient le temps de transport pour ce colis sur cet intervalle particulier. Le résultat final et un flux de tuples, chacun contenant les extrémités d'un intervalle et un temps de transport.

L'étape suivante est de comparer, pour un intervalle donné, la durée moyenne de transport des *m* colis précédents à la durée moyenne des *n* colis qui les précèdent sur le même intervalle.

32. En utilisant la palette BeepBeep Mtnp, cela revient à placer un unique processeur `Plot` à la fin de la chaîne actuelle.

La chaîne de processeurs réalisant cette opération est montrée dans la partie droite de la Figure 4.10.

Premièrement, le flux d'entrée est séparé selon la valeur de l'attribut *hop* (boîte #7) ; dans chaque tranche, la valeur de l'attribut *dur* est extraite (boîte #1). Puis, le flux d'événements est envoyé dans une opération sur une *fenêtre coulissante* (boîte #2). Celle-ci requiert deux paramètres : la taille de la fenêtre (appelée n), et le processeur à appliquer sur chaque fenêtre, représenté par la boîte #5. Dans ce cas particulier, l'opération à réaliser est la moyenne glissante sur tous les événements de la fenêtre. Un calcul similaire est réalisé sur la seconde copie du flux pour une fenêtre de taille m (boîte #4) ; par la présence du processeur `Trim` (bloc #3), ces deux fenêtres sont déphasées de n événements. L'effet produit est que les processeurs #2 et #4 calculent une durée moyenne sur deux fenêtres : une sur le « présent » (W_1), et l'autre sur le « passé » (W_2).

Ces moyennes glissantes sur deux fenêtres peuvent être vues comme des « tendances », $\overline{w}_1$ et $\overline{w}_2$. La dernière étape est de comparer les moyennes glissantes sur ces deux fenêtres et de s'assurer qu'elles ne « dévient » pas trop. C'est la tâche du processeur #6, qui calcule le ratio $\overline{w}_1/\overline{w}_2$, et vérifie que le résultat n'excède par une certaine valeur k prédéfinie. Le résultat est un flux de booléens ; la valeur $\perp$ est émise si et seulement si, pour un intervalle donné, le temps moyen de transport des m derniers colis excède par un facteur k le temps moyen de transport des n colis les précédant. Le processeur #8 réalise la conjonction de ces valeurs pour tous les intervalles.

Cette structure diffère ce celles vues précédemment sur plus aspects. Premièrement, une erreur n'est pas causée par un seul événement, mais par la manifestation d'une déviation d'une tendance calculée sur de multiples événements. Deuxièmement, ce qui est considéré « incorrect » est relatif à la fenêtre des événements passés calculée sur le *même* flux. Enfin, il

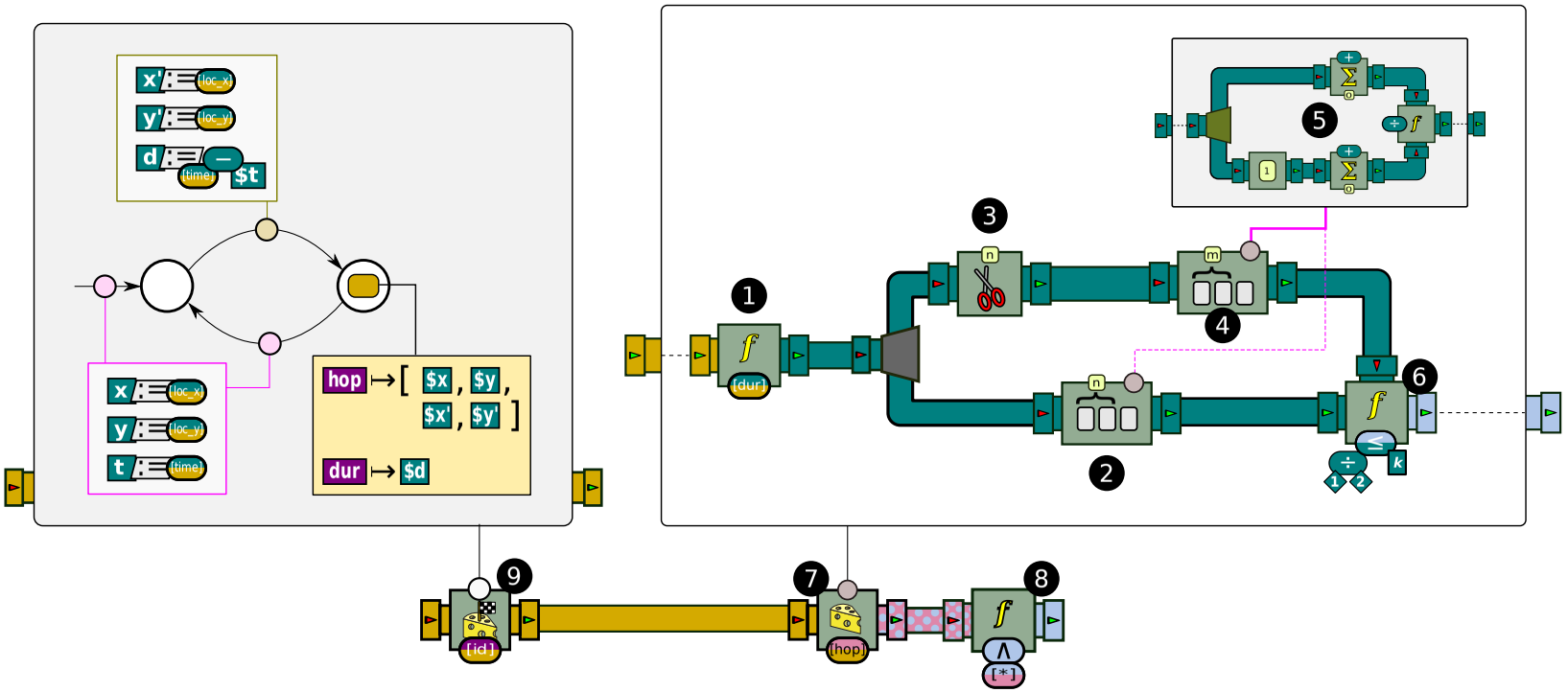


FIGURE 4.10 : La chaîne de processeurs pour la Propriété 10 (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

faut noter que chaque segment entre deux points est considéré de manière isolée : le temps moyen de transport va généralement différer d'un segment à un autre, donc chacun d'entre-eux est comparé à son propre passé.

Ce modèle est en réalité générique. En remplaçant le processeur #5 par un autre, et en changeant les fonctions de distance de la boîte #6 en conséquence, il est possible de calculer de nombreuses tendances sur un flux d'événements, et de détecter les déviations de ces tendances en temps réel. Ces tendances n'ont même pas à être des scalaires, en supposant qu'une fonction de distance métrique appropriée est fournie pour pouvoir les comparer. À cet égard, la Figure 4.10 est une instance de modèle de *distance de tendances auto-corrélée*, un concept introduit précédemment dans une étude de forage de données sur les flux d'événements réalisée par Roudjane *et al.* (2018).

CHAPITRE V
LES SÉQUENCES DE PAIR-ACTIONS

Dans le chapitre précédent, nous avons vu comment la blockchain et les *smart contracts* pouvaient être utilisés pour permettre à une multitude d'acteurs d'interagir avec un historique partagé et authentifié à travers un système de suivi de colis. Chacun des acteurs possède alors une copie de cet historique, qui concerne la totalité des colis traversant leur réseau. Cependant, cette situation présente des incohérences majeures. En effet, un acteur qui n'est pas concerné par un colis (*p. ex.*, un livreur ou un *hub* n'ayant jamais interagi avec un colis spécifique) en conserve tout de même l'historique. Cela, en plus d'être inutile, nécessite tout de même que chaque agent stocke ces informations potentiellement superflues. Étant donné que la blockchain est persistante et immuable, cela signifie que toujours plus de données vont s'amasser, alors qu'une infime partie d'entre elles seront pertinentes pour un acteur spécifique.

Il est important de noter que cette observation est valide même si l'historique n'est pas conservé par tous les acteurs, mais seulement par certains. Si, par exemple, un client Ethereum est présent dans chaque *hub* (ce qui serait sans doute plus réaliste) et que les livreurs se connectent à l'un d'eux pour stocker leurs interactions avec les colis, cela n'empêche pas que la majorité des *hubs* ne traitera probablement jamais certains colis et en conservera tout de même les informations inutilement.

Ainsi, un agent ne devrait avoir accès qu'aux historiques des colis qu'il a traité ou qu'il traite actuellement. On peut alors imaginer que le colis en lui-même stocke son historique et que chaque acteur interagisse avec celui-là lorsqu'il manipule directement le colis. Par exemple, lorsque le livreur ramasse un colis dans un *hub*, il l'écrit dans l'historique ; lorsqu'il le déposera à un autre *hub*, ce dernier pourra alors consulter son historique, y écrire que le

colis a bien été reçu, et potentiellement synchroniser le tout sur un serveur distant. Un autre livreur pourra alors récupérer le colis, interagir avec son historique et ainsi de suite jusqu'à sa livraison finale.

À cet égard, utiliser la blockchain nous semble être une solution quelque peu excessive. En effet, certains mécanismes intrinsèques aux blockchains populaires telles qu'Ethereum entraînent des lourdeurs qui ne sont pas toujours nécessaires. Par exemple, dans certains contextes où les réseaux de blockchain sont privés (comme ce peut être le cas dans la logistique ou les services médicaux), les notions de *gas* ou de prix des transactions ont peu d'utilité. De plus, les consensus représentent encore aujourd'hui à la fois la force *et* la limite des blockchains, permettant des taux de transactions assez faibles et, dans le cas du Proof-of-Work, nécessitant de lourds calculs pour ajouter des blocs.

Ces éléments sont à mettre en parallèle avec ce que la blockchain *ne permet pas*, tout du moins pas de façon native. Le fait que la blockchain soit ouverte et son contenu lisible par tous est certes un avantage dans certains contextes, mais l'absence totale de confidentialité des actions peut être un vrai manque dans d'autres cas. En effet, dans l'exemple des dossiers médicaux vu en Section 1.1, il est nécessaire que les informations médicales des patients restent confidentielles. En outre, certaines données ou actions doivent être accessibles à des personnes spécifiques et pas à d'autres, faisant intervenir alors les notions de groupes et de contrôle d'accès qui, là encore, ne sont pas présentes dans la blockchain. Ces fonctionnalités *peuvent* certes être mises en place dans une blockchain par le biais de *smart contracts*, mais cela amène nécessairement aux lourdeurs citées plus tôt et à d'autres, comme le besoin de développer des *smart contracts* spécifiques à chaque utilisation.

Par conséquent, dans ce chapitre nous décrivons une nouvelle approche, la séquence de pair-actions, permettant de construire des historiques sécurisés. Cette approche s'acquitte

du besoin de minage ou de consensus. Plutôt, sa sécurité est directement assurée par le chiffrement : les actions y sont à la fois signées par leur auteur et chiffrées pour le groupe d'utilisateurs au nom duquel elles sont réalisées. Plus légère et flexible que la blockchain traditionnelle, la séquence de pair-actions permet donc de fournir un historique immuable, confidentiel et authentique que les *pairs* (c.-à-d., les différents individus impliqués dans sa manipulation) vont pouvoir utiliser pour vérifier la conformité du cycle de vie de l'artefact. Ce dernier sera référé par le terme de *document*, qui désignera un type d'artefact particulier correspondant à la structure en Section 5.1.1. Cette définition peut d'ailleurs être étendue à certains objets physiques, comme les cartes MIFARE (vues en Section 1.3) ou les tags RFID (pouvant, par exemple, être placés à l'intérieur de conteneurs ou de plus petits colis).

5.1 FORMALISATION DÉTAILLÉE DES CYCLES DE VIE

Dans cette section, nous formalisons les notions qui seront manipulées tout au long de ce chapitre, à savoir les documents, les pair-actions, les actions, et les cycles de vie.

Nous y supposons l'existence d'une fonction de hachage $\hbar$. Pour simplifier la notation, nous supposons que le codomaine de $\hbar$ est $\mathbb{H}$. Nous définissons P comme étant l'ensemble des pairs, c.-à-d. les individus ou entités impliqués dans la manipulation du document. Les pairs appartiennent à un ou plusieurs groupes, qui sont comparables à la notion de rôle dans les modèles de contrôle d'accès classiques tels que RBAC (Ferraiolo & Kuhn, 1992; Sandhu *et al.*, 1994). On note l'ensemble des groupes G ; il contient les étiquettes identifiant chaque groupe. Nous verrons que l'appartenance à un groupe donne des accès en lecture/écriture à un certain nombre de champs du document considéré (voir Section 5.1.3).

5.1.1 DOCUMENTS

Soit D un ensemble de documents. Un document $d \in D$ est un ensemble composé des trois types d'éléments suivants :

— Valeurs : une valeur est une donnée typée qui est référencée par un identifiant unique au sein de d. Cet identifiant est un appelé une *clé*.

— Listes : une liste est un tableau dans lequel chaque élément peut désigner un autre document, à savoir une valeur, une autre liste ou une *mappe*.

— Mappes : une mappe est un ensemble de paires clé-document, où chaque clé désigne un autre document, à savoir une valeur, une liste ou une autre mappe.

Soit $\mathscr{V}$ un ensemble de valeurs, $\mathscr{K}$ un ensemble de clés, $\mathscr{L}$ un ensemble de listes, et $\mathscr{M}$ un ensemble de mappes. Selon les précédentes définitions, nous pouvons représenter une liste $l \in \mathscr{L}$ comme une fonction $l : \mathbb{N} \to D$ et une mappe $m \in \mathscr{M}$ comme une fonction $m : \mathscr{K} \to D$. Par conséquent, nous définissons D comme un sous ensemble de valeurs, de listes et de mappes : $D \subseteq \mathscr{V} \cup \mathscr{L} \cup \mathscr{M}$. Un document spécial, noté $d_\emptyset$, sera appelé le *document vide*.

ACCÈS AUX ÉLÉMENTS

Cette sous-section vise à définir un moyen d'accéder à un élément spécifique, qui est lui-même considéré comme un autre document, à l'intérieur d'un document d. Pour cela, nous définissons une fonction de *chemin* π. Soit « * » l'opérateur définissant une *séquence*, on note $\pi : \mathscr{P}^* \times D \to D$, où $\mathscr{P}$ est un ensemble *d'éléments de chemin*, c.-à-d. des entiers ou des clés, $\mathscr{P} \subseteq \mathbb{N} \cup \mathscr{K}$. En un mot, π prend une séquence d'éléments de chemin et un document en entrée, et retourne le sous-document correspondant, supposant que ce dernier existe et

que le chemin donné est correct. Formellement, nous pouvons identifier quatre expressions différentes pour π dépendant de la nature des entrées.

Définition 7 (Fonction de chemin)**.** *Soient $l \in \mathscr{L}$ une liste, $m \in \mathscr{M}$ une mappe, et $k \in \mathscr{K}$ une clé de m, on note respectivement $l(n)$ ($n \in \mathbb{N}$) le n-ième document de l, et $m(k)$ le document assigné à la clé k dans m. Soit $\overline{\sigma} \in \mathscr{P}^*$, une séquence finie d'éléments de chemin. Nous écrivons $\overline{\sigma}' \prec \overline{\sigma}$ pour indiquer que $\overline{\sigma}'$ est un préfixe de $\overline{\sigma}$. Soit $\varepsilon \in \mathscr{P}$ le chemin vide, alors la fonction de chemin π est définie comme suit :*

$$\pi(\overline{\sigma}, d) \triangleq \begin{cases} \pi(\overline{\sigma}', \pi(n, l)) = \pi(\overline{\sigma}', l(n)) & n \in \mathbb{N}, l \in \mathscr{L} \quad (5.1) \\ \pi(\overline{\sigma}', \pi(k, m)) = \pi(\overline{\sigma}', m(k)) & k \in \mathscr{K}, m \in \mathscr{M} \quad (5.2) \\ \pi(\varepsilon, d) = d & (5.3) \\ d_\emptyset & \textit{sinon} \quad (5.4) \end{cases}$$

Dans le cas où d est une liste l, soit n un entier naturel, alors le chemin cherché réside dans le n-ième élément de l (cas 5.1). Si d est une mappe et k une clé, le chemin cherché est dans le document associé à la clé k (cas 5.2). En bouclant sur ces deux cas, quand le chemin devient finalement vide (ce qui veut dire que le document ciblé a été trouvé), π retourne le document correspondant (cas 5.3). Cependant, si à un moment donné le chemin d'entrée est invalide ou n'existe pas dans le document fourni, *p. ex.* k n'existe pas dans m ou n est un index invalide de l, la fonction π retournera le document vide $d_\emptyset$.

```
{
  "a": {
    "b": [0, 1, 6],
    "c": 9
  },
  "d": 5
}
```

FIGURE 5.1 : Exemple de document JSON (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

FORMATS DE FICHIER EXISTANTS

Plusieurs formats de fichier satisfont notre représentation des documents et notre fonction de chemin. Évidemment, le JSON, pour JavaScript Object Notation (ECMA International, 2013), est l'un d'entre eux puisqu'il consiste justement en une écriture sous la forme de paires attributs-valeurs. En utilisant des expressions XPath (W3C, 1999) comme des éléments de chemin, XML peut donc être aussi une bonne solution. Cependant, des documents encore plus humainement intelligibles peuvent correspondre, comme les fichiers PDF. En effet, en utilisant des bibliothèques courantes (telles que PDFBox [33] ou PDFtk [34]), il est possible d'assigner un nom unique à un objet à la création d'un nouveau PDF. Par exemple, une unique chaîne de caractères, utilisée comme *clé*, peut être associée à un champ de texte. Ce faisant, il est possible de récupérer la valeur du champ en utilisant le nom spécifié qui, dans notre cas, sera considéré comme le chemin menant à cette valeur au sein du document.

Pour plus de clarté, considérons le document JSON d décrit dans la Figure 5.1. Conformément à la syntaxe du JSON, les accolades « `{}` » représentent une mappe et les crochets « `[]` » une liste ou un tableau. La fonction de chemin π peut être utilisée pour atteindre dif-

33. https://pdfbox.apache.org/

34. https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/

férentes sections du document. Par exemple, en supposant que le caractère « / » sépare les éléments de chemin, $\pi(\text{a/b/2}, d) = 6$ puisque cela correspond à l'accès à l'élément « a » (une mappe), puis à l'élément « b » dans cette mappe (une liste), et enfin au troisième élément de cette liste (en supposant que le premier indice de la liste est 0).

5.1.2 ACTIONS

Soit D un ensemble de documents et A un ensemble d'actions. Chaque action $\mathtt{a} \in A$ est associée à une fonction $f_{\mathtt{a}} : D \to D$ qui prend un document en entrée, et retourne un autre document.

Soit AT un ensemble de types d'action, tels que `add`, `delete`, `read` que nous définissons plus tard. Formellement, une action `a` est un triplet $\langle \bar{\sigma}, t, v \rangle \in (\mathscr{P}^* \times AT \times \mathscr{V})$ où $\bar{\sigma}$ est le chemin menant au document sur lequel l'action est réalisée, t le type d'action exécutée, et v la valeur à associer au document ciblé. Nous introduisons les notations $\mathtt{a}.\bar{\sigma}$, $\mathtt{a}.t$, et $\mathtt{a}.v$ pour référencer, respectivement, le chemin, le type, et la valeur de l'action `a`. Notons que, pour certaines actions, v peut être vide, puisque certains types d'actions ne nécessitent aucune valeur. Par exemple, une action `delete` pourrait n'avoir aucune valeur puisqu'elle effacerait simplement la valeur actuelle sans la remplacer par une autre. Cela est également le cas de l'action `read` qui ne devrait modifier aucune valeur.

Nous utilisons la notation suivante pour représenter les actions : $\bar{\sigma}(\mathtt{type}, \mathtt{data})$, où $\bar{\sigma}$ est le chemin du document ciblé, `type` est le type de l'action, et `data` est la nouvelle valeur si nécessaire. On peut alors écrire $\mathtt{patient\_number}(\mathtt{write}, \mathtt{1A7452N})$ pour représenter l'action qui assigne la valeur `1A7452N` au numéro du patient ; ici `patient_number` correspond au chemin au sein du document qui amène au champ correspondant. De la même façon, écrire $\mathtt{side\_effects}(\mathtt{add}, \mathtt{Nausea})$ indique que la valeur `Nausea` devrait être ajoutée au paramètre

`side_effects`, qui est une liste. Remplacer le quatrième élément de cette même liste avec la valeur `Headache` serait écrit $\texttt{side\_effects}/3(\texttt{overwrite}, \texttt{Headache})$. À cet égard, nous pouvons distinguer deux catégories globales d'action, décrites dans les deux sous-sections suivantes.

ACTIONS D'ALTÉRATION

Une action d'altération est une action qui vise à modifier concrètement le contenu et les valeurs d'un document. Soit $(d, d') \in D^2$, une action d'altération `aa` peut être associée à la fonction $f_{\texttt{aa}}$ telle que $f_{\texttt{aa}}(d) = d'$ où $d' \neq d$. Cela signifie que le document résultant est strictement différent du document originel.

Par exemple, cette catégorie d'actions pourrait inclure les actions de type `add`, `delete` ou `overwrite`. Une action `add` pourrait désigner le fait d'ajouter une nouvelle clé ou une valeur à un document, l'action `delete` supprimer une valeur pour un document spécifié, et `overwrite` modifier le contenu d'un document existant.

Tandis que ces actions mènent à des modifications évidentes, il est possible qu'un pair réalise une action qui devrait modifier le document, une action d'altération, mais dont la nouvelle valeur est la même que la précédente (*p. ex.*, remplacer la valeur d'un document avec exactement la même valeur que l'actuel). Cela résulterait en un document identique et signifierait que l'action n'est pas une action d'altération (conformément à leur définition formelle). Néanmoins, ce type d'action étant peu pertinente (puisqu'elle n'ajoute aucune information), nous ne les prendrons pas en compte.

ACTIONS D'OBSERVATION

Certaines actions, par opposition, peuvent fournir une information sans pour autant altérer le contenu du document. Ce type d'action, que nous appelons les actions d'observation, indique que quelque chose a été fait sur ou avec le document. Par conséquent, soit $d \in D$, une action d'observation `oa` peut être associée à une fonction $f_{\texttt{oa}}$ telle que $f_{\texttt{oa}}(d) = d$.

Par exemple, tout effet secondaire relatif à un médicament prescrit qui a été rapporté par le patient devrait être observé ou au moins porté à la connaissance du médecin traitant. Dans ce cas, observer un effet secondaire ne devrait rien modifier au contenu du document, mais les pairs souhaitant contrôler s'il a bien été approuvé devraient en être capables.

Une des manières possibles pour implémenter ce type d'action serait d'ajouter un champ contenant un booléen qui serait rempli avec un « TRUE » quand le médecin approuve l'effet. Ceci fonctionnerait pour des actions de type *approbation* mais cela rendrait d'autres actions de lecture/accès peu pratiques à implémenter. Dans le pire des scénarios, il pourrait être nécessaire d'enregistrer *tous* les accès au document, et ajouter des champs supplémentaires pour garder cette information complexifierait grandement la structure.

En outre, les actions d'observation nous permettent de traiter ce type d'actions sans modifier la structure du document. Ces accès sont simplement enregistrés dans l'historique des actions, sans aucune modification du contenu du document en lui-même. En d'autres termes, les actions d'observation sont comparables à des « tampons » ajoutés à la séquence d'actions pour signaler le fait qu'un certain pair a vu ou approuvé un champ d'un document dans son état actuel.

5.1.3 CYCLES DE VIE

Avant de définir en détail ce qu'est un cycle de vie, il est nécessaire d'introduire deux autres éléments : les pair-actions et les séquences de pair-actions. Une pair-action est un quadruplet $\langle \mathtt{a}, p, g, h \rangle \in (A \times P \times G \times \mathbb{H})$ contenant une action $\mathtt{a}$, un identifiant du pair responsable de cette action p, et g, le groupe au nom duquel l'action est prise (l'utilisation de la valeur de hachage h sera expliquée plus tard). Nous construisons une séquence de pair-actions et la notons $\bar{s}$. L'ensemble S désigne toutes les séquences de pair-actions possibles.

Le cycle de vie d'un document spécifie quelles sont les actions que les pairs sont autorisés à effectuer et dans quel ordre. Il est représenté par une fonction $\delta : S \to \{\top, \bot\}$. Intuitivement, la fonction prend en entrée une séquence de pair-actions et décide si elle est valide ($\top$) ou non ($\bot$). Comme nous l'avons montré en Chapitre 1, un cycle de vie peut être vu comme des ensembles de contraintes que les séquences de pair-actions doivent respecter. Nous classifions ces contraintes en trois catégories : les contraintes d'accès, d'intégrité, et d'ordonnancement. Ces catégories correspondent aux exemples de cycles de vie vus en Chapitre 1, notamment ceux portant sur la gestion de dossiers médicaux et de suivi de colis dans une logistique hyperconnectée.

CONTRAINTES D'ACCÈS

Puisque les champs de compétences et les responsabilités peuvent être divers et variés au sein d'une entreprise ou d'une institution, il semble logique qu'il puisse y avoir au moins autant de niveaux d'autorisation et de règles de contrôle d'accès pour un document. Par exemple, un ingénieur pourrait être autorisé à remplir la partie technique d'un projet, mais seul le gestionnaire du projet pourrait signer le document. Aussi, l'ingénieur pourrait ne pas avoir

accès aux détails financiers du projet, tandis que le gestionnaire aurait accès au document en entier. Ces exemples font partie des contraintes d'accès.

Comme il l'a été dit dans la Section 5.1.3, un pair réalise une action toujours au nom d'un groupe, ce qui requiert évidemment qu'il fasse lui-même partie du groupe en question. Ceci signifie que les pairs ont différents droits selon le groupe au nom duquel ils effectuent l'action. Par conséquent, pour déterminer si un pair $p \in P$ est autorisé à effectuer une action $\mathtt{a} \in A$ au nom du groupe $g \in G$, il faut d'abord vérifier si p appartient réellement au groupe g et si les membres de g sont bien autorisés à exécuter `a`. L'appartenance au groupe peut être évaluée en utilisant le prédicat $M : P \times G \to \{\top, \bot\}$, où $M(p,g) = \top$ signifie que le pair p appartient bien au groupe g, et $M(p,g) = \bot$ lorsque ce n'est pas le cas. L'autorisation d'un groupe à effectuer une action est évaluée par la fonction $access : G \times A \to \{\top, \bot\}$, où $access(g, \mathtt{a}) = \top$ signifie que les membres de g sont autorisés à accomplir `a`, et $access(g, \mathtt{a}) = \bot$ lorsqu'ils ne le sont pas.

Pour chaque groupe $g \in G$, nous attribuons une fonction de cycle de vie de groupe δ_g, et nous associons les pairs aux groupes en utilisant le prédicat $M(p,g)$. La fonction δ_g spécifie les actions réalisables sur le document par un membre du groupe g. Un pair $p \in P$ appartient à l'ensemble de groupes $G_p = \{g \mid M(p,g) = \top\}$. Le cycle de vie que p vérifiera est $\delta_p(s) = \bigwedge_{g \in G_p}(\delta_g(s))$, où $\top$ et $\bot$ sont interprétés respectivement comme les booléens `true` et `false`. Le cycle de vie δ_p assure que p peut uniquement vérifier les cycles de vie des groupes auxquels il appartient. Nous ajoutons la restriction que quand un pair exécute une action sur un document, il l'effectue toujours au nom d'un seul groupe. Dans ce cas, il est à noter que le cycle de vie du groupe g, δ_g, agit sur la séquence entière. De fait, la spécification doit donc être écrite de façon à que δ_g soit uniquement concerné par les actions pertinentes pour le groupe, ignorant le reste de la séquence.

Évidemment, les pairs peuvent potentiellement faire partie de plusieurs groupes différents. Les groupes et leurs accès pourraient être gérés en utilisant des mécanismes tels que le *Role-Based Access Control*, abrégé RBAC (Ferraiolo & Kuhn, 1992; Sandhu *et al.*, 1994). Un *rôle* serait alors associé à chaque pair, et celui-là serait associé à plusieurs groupes. Par exemple, le rôle d'ingénieur serait associé aux groupes *Ingénieurs* et *Employés*, le rôle de comptable aux groupes *Comptables* et *Employés*, et le gestionnaire du projet serait associé aux groupes *Gestionnaires*, *Ingénieurs*, *Comptables* et *Employés*. Cela veut dire que le gestionnaire aurait alors les accès combinés des ingénieurs et des comptables, en plus de ses propres droits.

D'autres mécanismes tels que l'*Attribute-Based Access Control*, abrégé ABAC (Yuan & Tong, 2005), pourraient être utilisés. Au lieu d'associer des rôles existants à des groupes, l'appartenance à un groupe pourrait être déterminée par des attributs propres aux pairs tels que leur poste, leur département, ou encore les projets qui leur sont actuellement attribués.

CONTRAINTES D'INTÉGRITÉ

Les documents peuvent également avoir des contraintes internes sur leur contenu qui pourraient être basées sur son *format*. Par exemple, il peut être nécessaire qu'un champ dans un fichier PDF contienne uniquement du texte et que sa taille soit limitée, tandis qu'un attribut dans un objet JSON serait limité à contenir un tableau d'entiers exclusivement. De plus, les valeurs des documents pourraient être reliées entre elles sous certaines conditions. Par exemple, un document pourrait contenir un champ *prix du médicament* indiquant le prix d'un médicament acheté par un patient, et un champ *montant remboursé* qui indiquerait le montant remboursé sur l'achat du médicament par une compagnie d'assurance. Dans ce cas, la valeur du *montant remboursé* ne pourrait être plus grande que celle du *prix du médicament* ou d'un certain pourcentage de sa valeur.

Plus généralement, les contraintes d'intégrité sont des restrictions sur les champs des documents et définissent un sous-ensemble de documents valides $D_{\text{valid}} \subseteq D$. Soit un document $d \in D$ et une action $\mathtt{a} \in A$; contrôler l'intégrité de d après l'application de $\mathtt{a}$ revient à calculer $d' = f_{\mathtt{a}}(d)$ et à vérifier que $d' \in D_{\text{valid}}$. Si $d' \notin D_{\text{valid}}$, alors l'action viole les contraintes d'intégrité. Le contrôle d'une séquence entière d'actions est réalisé par le contrôle successif à chaque application d'une action, en s'assurant que le document reste valide.

Afin d'exprimer et d'appliquer de telles contraintes, nous pouvons nous inspirer de solutions existantes telles que le XACML (OASIS Standard, 2013) ou le WSDL (W3C, 2007). Ces deux langages basés sur le XML permettent de décrire des *politiques*, c.-à-d. une gestion de réponses/requêtes pour le XACML et la définition de services web pour le WSDL, et pourraient être adaptés pour correspondre à notre mécanisme. Cependant, deux autres langages, là aussi basés sur le XML, semblent parfaitement convenir à notre situation : le XML DTD (W3C, 2006) et le XML Schema (W3C, 2012). Le JSON Schema (Wright *et al.*, 2019) est aussi un bon candidat puisqu'il s'agit presque uniquement d'une adaptation du XML Schema au format JSON, c'est pour ce cela que nous n'en parlerons pas ici.

XML Document Type Definitions Un fichier DTD (Document Type Definition) permet de décrire une structure attendue pour un document XML. Le DTD offre une variété de possibilités pour exprimer des contraintes de format. Par exemple, nous avons créé le fichier `drug_prescription.dtd`, présenté en Figure 5.2, contenant une structure simple mais concrète pour une prescription de médicament.

La première ligne stipule qu'un élément appelé `drug_description` doit être composé d'éléments nommés `date`, `doctor_name`, `patient_nb`, et `side_effects`. En plus, la seconde ligne spécifie que `date` est composé d'une chaîne de caractères (`#PCDATA`), et ainsi

```
<!ELEMENT drug_prescription (date,doctor_name,patient_nb,side_effects)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT doctor_name (#PCDATA)>
<!ELEMENT patient_nb (#PCDATA)>
<!ELEMENT side_effects (side_effect+)>
<!ELEMENT side_effect (type,hours_since_prescription)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT hours_since_prescription (#PCDATA)>
```

FIGURE 5.2 : Fichier DTD `drug_prescription.dtd` d'une prescription de médicament (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

de suite pour les lignes suivantes. La combinaison de ces déclarations décrit précisément la structure possible du document. Tout document respectant ce format est une *prescription de médicament* valide telle que définie dans le fichier DTD. Par exemple, le fichier décrit en Figure 5.3 est valide vis-à-vis du fichier `drug_prescription.dtd`.

XML Schema Le XML Schema, aussi appelé XML Schema Definition (XSD), est très similaire au DTD. D'ailleurs, il est même possible de croiser des fichiers XSD et des fichiers DTD. Cependant, avec le XSD, il est possible de spécifier des types aux éléments (*p. ex.*, booléen, date, entier) ainsi que des restrictions telles que l'appartenance à une énumération (c.-à-d., la valeur doit résider parmi une liste de valeurs possibles) ou le respect d'un format précis (c.-à-d., la valeur doit respecter une expression régulière). La même structure de prescription de médicament décrite dans le fichier `drug_prescription.dtd` peut être cette fois spécifiée en XSD dans la Figure 5.4 ; l'exemple de document XML donné en Figure 5.3 est aussi valide selon cette définition.

```xml
<?xml version="1.0"?>
<!DOCTYPE drug_prescription SYSTEM "drug_prescription.dtd">
<drug_prescription>
  <date>2017-05-02</date>
  <doctor_name>Dr. Meredith Grey</doctor_name>
  <patient_nb>1A7452N</patient_nb>
  <side_effects>
    <side_effect>
      <type>Headache</type>
      <hours_since_prescription>2</hours_since_prescription>
    </side_effect>
    <side_effect>
      <type>Nausea</type>
      <hours_since_prescription>3</hours_since_prescription>
    </side_effect>
  </side_effects>
</drug_prescription>
```

FIGURE 5.3 : Exemple de fichier XML respectant le DTD `drug_prescription.dtd` (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

CONTRAINTES D'ORDONNANCEMENT

Il pourrait être attendu que les actions réalisées sur un document respectent un certain ordre chronologique. Par exemple, si un document doit être signé, il ne devrait l'être que lorsque toutes ses sections ont été remplies. De plus, le traitement des différentes sections pourrait également être amené à respecter une séquence spécifique, puisque certaines sections pourraient se référer à d'autres. Quelle que soit la raison, les contraintes d'ordonnancement sont largement utilisées, notamment dans le traitement de processus opérationnels, et forment la base de ce que sont les cycles de vie d'artefacts.

```
<xs:element name="side_effect">
  <xs:complexType>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="hours_since_prescription"
        type="xs:integer"/>
  </xs:complexType>
</xs:element>

<xs:element name="side_effects">
  <xs:complexType>
    <xs:element ref="side_effect" maxOccurs="unbounded"/>
  </xs:complexType>
</xs:element>

<xs:element name="drug_prescription">
  <xs:complexType>
    <xs:element name="date" type="xs:date"/>
    <xs:element name="doctor_name" type="xs:string"/>
    <xs:element name="patient_nb" type="xs:string"/>
    <xs:element ref="side_effects"/>
  </xs:complexType>
</xs:element>
```

FIGURE 5.4 : Fichier XSD décrivant le même format de prescription de médicament que `drug_prescription.dtd` (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

De telles contraintes peuvent être spécifiées de différentes manières. Le Chapitre 2 décrit différentes méthodes existantes telles que les automates finis (FSM), le paradigme du GSM ou le BPMN. Cependant, les réseaux de Petri, la Logique Temporelle Linéaire (LTL), ou tout autre modèle ou langage décrivant une séquence prédéterminée d'actions peut être utilisé. Pour adapter les contraintes d'ordre à notre formalisation, nous notons que la vérification de la conformité d'une pair-action par rapport à des contraintes d'ordre ne s'appuie que sur l'action réalisée, et non le pair ou le groupe (voir Section 5.1.3 pour une discussion de cette problématique). Plus précisément encore, seuls le type et le document ciblé par l'action impliquée sont décisifs. Soit $\overline{\mathtt{a}} = (\mathtt{a_0}, \mathtt{a_1}, \ldots, \mathtt{a_n})$ une séquence finie d'actions, et $\mathtt{a_{n+1}}$ une nouvelle action. La fonction $order : A^* \times A \to \{\top, \bot\}$ évalue si cette nouvelle action respecte les contraintes d'ordre par rapport aux anciennes actions ($order(\overline{\mathtt{a}}, \mathtt{a_{n+1}}) = \top$), ou pas ($order(\overline{\mathtt{a}}, \mathtt{a_{n+1}}) = \bot$).

Dans les prochains paragraphes, nous montrons comment nous pouvons adapter deux des méthodes précédemment mentionnées pour exprimer des contraintes d'ordonnancement.

Automates finis Les automates finis, ou machines à états finis, sont des modèles mathématiques permettant de caractériser l'évolution du comportement d'un système lorsque ce dernier est soumis à des *entrées*. Nous avons présenté leur formalisation dans le Chapitre 2, et nous l'adaptons à notre cas. Ainsi, l'alphabet pourrait contenir seulement des chaînes de symboles, chacune représentant un couple $\langle \bar{\sigma}, type \rangle \in (\mathscr{P}^* \times AT)$ impliquant le document ciblé et le type de l'action. Soit $\mathscr{A}$ un automate fini, Σ son alphabet, Q son ensemble d'états, $q_0 \in Q$ son état initial et $q_{invalid} \in Q$ un état final. Nous notons $\varphi : A \to \Sigma$ la fonction qui prend une action en entrée et la transforme en un élément valide de l'alphabet, en se basant sur son type et le document qu'elle cible. En partant de q_0, nous traitons la toute première action $\mathtt{a_0}$: si $\varphi(\mathtt{a_0})$ correspond à une transition possible depuis q_0, alors l'action est valide

et nous procédons au prochain état ; sinon, le prochain état est $q_{invalid}$. Par conséquent, en traitant successivement les actions et en procédant d'états en états (valides ou invalides), nous pouvons déterminer si ces actions sont conformes ou pas. Cette formalisation nous permet d'établir un ordre séquentiel basé sur les actions réalisées et le document en question, ce qui est exactement ce que nous attendons des contraintes d'ordonnancement telles que nous les avons définies dans cette section. À noter que la même méthode peut être utilisée pour adapter les réseaux de Petri afin qu'ils puissent traiter des actions, la seule différence étant que la fonction φ devrait être $\varphi : A \rightarrow T$ où T est l'ensemble des transitions du réseau de Petri.

LTL Nous avons vu dans le Chapitre 2 que LTL est une logique temporelle modale qui peut être utilisée pour évaluer si une trace spécifique respecte certaines conditions en termes d'ordonnancement. Ces conditions sont constituées des opérateurs de logiques classiques et des opérateurs modaux temporels pour former des *formules* LTL. Soient $\overline{\mathsf{a}}$ une séquence d'actions et $\overline{\mathsf{a}}^i = (\mathsf{a}_i, \mathsf{a}_{i+1}, \ldots, \mathsf{a}_n)$ un suffixe de $\overline{\mathsf{a}}$. Le fait que $\overline{\mathsf{a}}$ satisfasse une formule donnée ψ est noté $\overline{\mathsf{a}} \models \psi$. Dans le cas présent, les termes clos d'une formule LTL sont des tuples de la forme $\langle \bar{\sigma}, type \rangle \in (\mathscr{P}^* \times AT)$. Un élément e d'une séquence satisfait le terme clos $\langle \bar{\sigma}, type \rangle$ si la cible de son action est $\bar{\sigma}$ et son type *type*. Nous pouvons adapter la sémantique de LTL à nos actions ; le résultat est montré dans la Figure 5.5.

5.2 APPLICATION DE CYCLE DE VIE PAR SÉQUENCE DE PAIR-ACTIONS

Pour pallier les problèmes mentionnés dans le Chapitre 2 et en introduction de ce chapitre, nous décrivons dans cette section une technique originale pour stocker un historique des modifications directement dans le document. Étant donné que l'authenticité de cet historique est garantie (par l'utilisation de hachage et de chiffrement), cette technique permet à tout pair

$$
\begin{aligned}
\bar{\mathsf{a}} \models \langle \bar{\sigma}', type \rangle &\equiv \bar{\mathsf{a}}[0].\bar{\sigma} = \bar{\sigma}' \text{ et } \bar{\mathsf{a}}[0].t = type \\
\bar{\mathsf{a}} \models \neg\varphi &\equiv \bar{\mathsf{a}} \not\models \varphi \\
\bar{\mathsf{a}} \models \varphi \wedge \psi &\equiv \bar{\mathsf{a}} \models \varphi \text{ et } \bar{\mathsf{a}} \models \psi \\
\bar{\mathsf{a}} \models \mathbf{G}\,\varphi &\equiv \bar{\mathsf{a}}^i \models \varphi \text{ pour tout } i \\
\bar{\mathsf{a}} \models \mathbf{F}\,\varphi &\equiv \bar{\mathsf{a}}^i \models \varphi \text{ pour un } i \\
\bar{\mathsf{a}} \models \mathbf{X}\,\varphi &\equiv \bar{\mathsf{a}}^1 \models \varphi \\
\bar{\mathsf{a}} \models \varphi\,\mathbf{U}\,\psi &\equiv \bar{\mathsf{a}} \models \psi, \text{ ou } \bar{\mathsf{a}} \models \varphi \text{ et } \bar{\mathsf{a}}^1 \models \varphi\,\mathbf{U}\,\psi
\end{aligned}
$$

FIGURE 5.5 : Sémantique de LTL adaptée aux séquences d'actions (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

.

de récupérer un document, contrôler son historique et vérifier qu'il respecte bien un cycle de vie spécifié, et ce à tout moment.

Dans cette partie, nous supposons l'existence de fonctions de chiffrement/déchiffrement par clé publique ; la notation $E[M,K]$ désigne le résultat du chiffrement du message M avec la clé K, tandis que $D[M,K]$ correspond à son déchiffrement. Chaque pair $p \in P$ possède une paire de clés de chiffrement publique/privée notées respectivement $K_{p,u}$ et $K_{p,v}$. Nous décentralisons la spécification en introduisant différents groupes et pour chaque groupe $g \in G$, nous considérons une clé de chiffrement symétrique S_g.

La Figure 5.6 illustre l'approche générale pour appliquer des cycles de vie. Nous avons déjà défini le cycle de vie δ (voir Section 5.1.3). Ici, premièrement, nous montrons comment la séquence peut être chiffrée pour cacher des actions provenant de divers groupes, et comment son *condensat* (une valeur de hachage qui *résume* l'historique d'action) est calculé afin d'assurer son intégrité. De plus, nous expliquons comment la séquence peut être vérifiée étant donné son condensat, puis déchiffrée et contrôlée par chaque pair p en se basant sur leurs permissions (δ_p, voir Section 5.1.3).

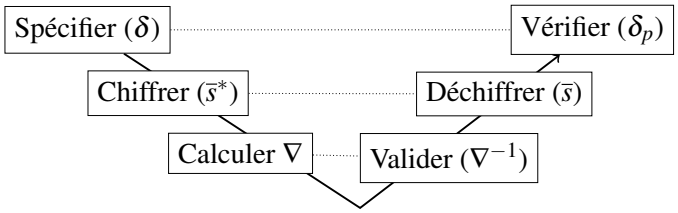


FIGURE 5.6 : Application du cycle de vie (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

5.2.1 CHIFFRER UNE SÉQUENCE

Avant de stocker une séquence de pair-actions dans le document, nous assurons la confidentialité des actions de groupe. Dans cette méthode, nous cherchons à interdire aux pairs qui ne sont pas membres d'un groupe donné de voir quelles actions exactes ont été effectuées au nom de celui-ci, mais pas le fait que des action aient été effectuées. Dans ce cas, une pair-action est un quadruplet $\langle \mathtt{a}, p, g, h \rangle$, signifiant que le pair p entreprend une action $\mathtt{a}$ au nom du groupe g ; l'élément h est un condensat, et son calcul est détaillé dans la sous-section suivante. Une pair-action est chiffrée comme suit : $pa^* = \langle E[\mathtt{a}, S_g], p, g, h \rangle$.

La séquence de pair-actions en question, chiffrée et stockée dans le document, est $\overline{s}^* \in (\mathbb{H} \times P \times G \times \mathbb{H})^*$. Ceci assure que les membres extérieurs au groupe ne puissent voir l'action effectuée au nom du groupe g ; ils sont donc capables de vérifier $M(p,g)$, mais ne peuvent voir quelle action ($\mathtt{a}$) a été réalisée. Ils ne peuvent donc pas savoir quelle modification $f_{\mathtt{a}}$ a été appliquée au document.

5.2.2 CALCULER LE CONDENSAT

L'authenticité et l'intégrité du sont garanties par le calcul et la manipulation du *condensat* de son historique.

Définition 8 (Condensat)**.** *Soit* $\overline{s}^* = (pa_1^*, \dots pa_n^*)$ *une séquence de pair-actions de taille n, et* $\overline{s'}^* = (pa_1^*, \dots, pa_{n-1}^*)$ *la même séquence de pair-actions (sans sa dernière pair-action), où* $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ *pour* $i \in [0,n]$*. Le condensat de* $\overline{s}^*$*, noté* $\nabla(\overline{s}^*)$*, est défini de la façon suivante :*

$$\nabla(\overline{s}^*) \triangleq \begin{cases} 0 & \text{si } n = 0 \\ E[\hbar(\nabla(\overline{s'}^*) \cdot a_n^* \cdot g_n), K_{v,p_i}] & \text{sinon} \end{cases}$$

Pour chaque pair-action pa_i^* *dans* $\overline{s}^*$*,* h_i *correspond au condensat calculé au moment de son ajout :* $h_0 = 0$ *et* $\forall i \geq 1, h_i = E[h_{i-1} \cdot a_i^* \cdot g_i), K_{v,p_n}]$*. Si au moment considéré* $\overline{s}^*$ *est de taille n, on a donc* $h_n = \nabla(\overline{s}^*)$*.*

En d'autres mots, pour calculer le n-ième condensat h_n d'une séquence chiffrée $\overline{s}^*$, le pair p_n, responsable de la dernière action $\mathtt{a_n}$ au nom du groupe g, prend le dernier condensat calculé h_{n-1}, chiffre $\mathtt{a_n}$ avec la clé de groupe S_g, concatène $h_{n-1} \cdot E[a_n, S_g] \cdot g$, en calcule la valeur de hachage, et chiffre le tout en utilisant sa clé privée K_{v,p_n}. Un effet secondaire de cette « signature » est qu'elle assure que le contenu devant être chiffré est de taille constante, et qu'elle ne va pas s'accroître avec l'ajout de nouvelles actions à l'historique du document. Signer avec l'identifiant du groupe joint à l'action est effectué pour assurer l'authenticité du groupe annoncé.

Le condensat est fonction de l'historique entier du document, depuis son état initial. De plus, chaque étape de l'historique est signée avec la clé privée du pair ayant effectué l'action. Il est important de noter que chiffrer chaque tuple de l'historique séparément ne serait pas suffisant. N'importe quel pair pourrait facilement supprimer toute pair-action de l'historique, et prétendre que certaines actions n'ont pas existé. Dans la même lignée, un pair pourrait remplacer n'importe quel élément de la séquence par un autre élément de la même séquence, perpétrant ainsi une forme spéciale d'attaque par rejeu (*replay attack*). Ajouter la

position de l'action dans le condensat ne serait pas suffisant non plus, puisque tout suffixe de la séquence pourrait tout de même être supprimé. De plus, avec ce mécanisme, falsifier un nouveau condensat nécessite de connaître les clés privées des autres pairs.

5.2.3 VALIDER UN CONDENSAT

En plus des données qu'il contient, un document doit aussi contenir la séquence de pair-actions chiffrée et le condensat correspondant. Vérifier que la séquence correspond au condensat est fait en vérifiant les appartenances aux groupes et en validant les condensats tout au long de la séquence.

Définition 9 (Valider le condensat)**.** *Soit une séquence de pair-actions chiffrée* $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ *de taille n, et un condensat c. Soit* $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ *la même séquence sans la dernière pair-action, et* $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ *pour* $i \in [0,n]$*. La séquence* $\overline{s}^*$ *vérifie c si et seulement si* $\nabla^{-1}(\overline{s}^*, c) = \top$*, où :*

$$\nabla^{-1}(\overline{s}^*, c) \triangleq \begin{cases} \nabla^{-1}(\overline{s'}^*, h_{n-1}) \wedge M(p_n, g_n) \wedge D[h_n, K_{p_n,u}] = \hbar(h_{n-1} \cdot a_n^* \cdot g_n) & si\ n > 0 \\ \bot & sinon \end{cases}$$

Pour valider le condensat, il faut d'abord le déchiffrer avec la clé publique de p_n. Cela donne la valeur hachée de la séquence. Il faut ensuite vérifier que l'action a_n^* et le groupe g_n sont authentiques en les hachant et en les comparant avec la valeur de hachage signée. Détecter une manipulation frauduleuse du condensat ou d'une séquence de pair-actions se fait de la façon suivante :

1. calculer $D[h_n, K_{p_n,u}]$ produit un résultat insensé, indiquant que la clé privée utilisée pour calculer ce condensat partiel est différente de celle annoncée ; l'authenticité du pair est donc compromise ;
2. calculer $D[h_n, K_{p_n,u}]$ produit un résultat différent de $\hbar(h_{n-1} \cdot a_n^* \cdot g_n)$, invalidant l'authenticité de l'action et du groupe annoncé ;
3. observer que p_n n'appartient pas à g_n.

Il est important de noter que, même si l'action est cachée au pair, celui-ci est toujours capable de vérifier, à tout le moins, que le pair p_n appartient bien à g_n et d'observer que p_n a effectué une action.

5.2.4 DÉCHIFFRER UNE SÉQUENCE

Une fois qu'un pair a validé l'authenticité d'une séquence, il peut ensuite la déchiffrer pour traiter les actions. Le déchiffrement d'une séquence de pair-actions chiffrée dépend de ce que peut voir le pair. La nouvelle séquence dépendra des groupes auxquels il appartient.

Définition 10 (Déchiffrer une séquence)**.** *Soit une séquence de pair-actions chiffrée* $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ *de taille n. Soit* $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ *la même séquence, sans sa dernière pair-action, et* $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ *pour* $i \in [0, n]$.

$$\mathrm{SD}(\overline{s}^*, p) \triangleq \begin{cases} \mathrm{SD}(\overline{s'}^*, p) \cdot pa_n & si\ M(p, g_n) \wedge n > 0 \\ \mathrm{SD}(\overline{s'}^*, p) & si\ \neg M(p, g_n) \wedge n > 0 \\ \varepsilon & sinon \end{cases}$$

où $pa_n = \left\langle D[a_n^*, K_{g_n}], p_n, g_n, h_n \right\rangle$ *et* ε *est la séquence vide.*

Dans le cas où le pair appartient au groupe annoncé ($M(p, g_n)$), la dernière action est déchiffrée avec la clé du groupe ($D[a_n^*, K_{g_n}]$) et le tuple résultant est inclus dans la séquence déchiffrée. Autrement, si le pair n'appartient pas au groupe ($\neg M(p, g_n)$), le tuple entier est rejeté de la séquence.

5.2.5 VÉRIFIER LE CYCLE DE VIE

Un pair p vérifie le cycle de vie du document à partir de ses groupes. Pour ce faire, il calcule d'abord $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p)$, et s'assure que $\delta_p(\overline{s}_p) = \top$. Dans ce cas, $\overline{s}_p$ est une séquence pouvant être déchiffrée par p à partir de ses groupes, tandis que δ_p est le cycle de vie que p peut vérifier [35].

5.2.6 CONTRÔLE DU DOCUMENT

Le but du condensat est de fournir au destinataire du document une garantie sur l'authenticité de la séquence de pair-actions qu'il contient. Cette séquence, en retour, peut être utilisée pour vérifier que le document passé est valide et n'a pas été manipulé entre temps.

Considérons la séquence de pair-actions déchiffrée par p et désignée par $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p)$ $= (\langle \mathtt{a}_0, p_0, g_0, h_0 \rangle, \ldots, \langle \mathtt{a}_k, p_k, g_k, h_k \rangle)$. Puisque la séquence de pair-actions peut omettre certaines parties chiffrées, on a donc $|\overline{s}_p| \leq |\overline{s}^*|$. En partant du document initial $d_\emptyset$, il est possible de calculer le nouveau document $d = f_{\mathtt{a}_k}(f_{\mathtt{a}_{k-1}} \cdots (f_{\mathtt{a}_1}(d_\emptyset)) \cdots)$, et le comparer avec le document transmis. En d'autres mots, il est possible pour un pair de « rejouer » la séquence d'actions complète, en partant du document vide, et de comparer le résultat de cette séquence avec le document actuel. Puisque certaines actions sont cachées au pair, il n'est pas possible de

35. Cette vérification est possible pour n'importe quelle implémentation de δ, nous fournissons une formalisation basée sur les automates dans la Section 5.3.1

reconstruire entièrement le document à moins que p n'appartienne à tous les groupes possibles (dans ce cas $\bar{s}_p = \bar{s}$). Cependant, il est possible de vérifier une partie du document si l'on considère les suppositions suivantes sur la spécification :

1. Les données dans le document sont partitionnées en des ensembles disjoints deux à deux $d = \bigcup_{g \in G}(d_g)$.
2. Pour chaque action a apparaissant dans le cycle de vie δ_g, $f_{\mathtt{a}}$ modifie les données dans un seul ensemble de données d_g ou ne modifie aucune donnée (c.-à-d., elle ne modifie pas le document).

Avec ces hypothèses, les membres du groupe peuvent rejouer seulement les données pertinentes pour le groupe. Les actions associées à des fonctions qui ne modifient pas le document (des actions d'observation, voir Section 5.1.2) pourraient être utilisées pour synchroniser plusieurs groupes. On pourrait considérer que chaque ensemble de champs est chiffré avec la clé de groupe, de façon à ce qu'il ne soit pas visible pour les autres groupes.

Il est important de noter que, dans certains cas, la connaissance de la séquence de pair-actions et du document vide suffit à reconstruire le document entier sans avoir besoin de le transférer. Dans de tels cas, échanger la séquence et le condensat sont les seuls pré-requis. Cependant, il existe une situation où ceci ne peut s'appliquer, par exemple lorsque le document est un objet physique qui doit être passé d'un pair à un autre (comme dans le cas de la carte de métro), ou lorsque les données sujettes aux modifications sont un sous-ensemble de toutes les données contenues dans le document.

5.3 GESTION DES GROUPES

Puisque les pairs peuvent appartenir à plusieurs groupes, les actions entreprises au nom d'un groupe peuvent également en intéresser d'autres. Par conséquent, elles devraient être

visibles pour les groupes d'intérêts ; les pairs de ces groupes (mais pas des autres) devraient être capables de les voir. Dans notre approche, pour partager une action $\mathtt{a}_{shared}$ avec n groupes, il est donc nécessaire de joindre l'action à la séquence de pair-actions n fois. Chaque instance est chiffrée avec la clé symétrique d'un des groupes. D'un côté, cela pourrait mener à une certaine inefficacité puisque l'action est dupliquée. D'un autre, cela force à poser des restrictions sur la fonction de vérification du cycle de vie δ. Si un pair p appartient à deux groupes ou plus partageant l'action $\mathtt{a}_{shared}$, alors il y aura respectivement deux répétitions ou plus de $\mathtt{a}_{shared}$ dans sa séquence déchiffrée. De fait, pour contrôler correctement δ, le cycle de vie doit prendre en compte les répétitions, et son application devrait être insensible à ces répétitions (elle doit être *stutter-invariant* (Wilke, 1999)).

Pour éviter ce problème, nous recommandons plutôt d'utiliser de nouveaux groupes pour les actions partagées entre plusieurs groupes. Nous appelons ces groupes des super-groupes. Nous créons un nouveau groupe g_{super}, avec tous les pairs impliqués, et chiffrons $\mathtt{a}_{shared}$ avec $K_{g_{super}}$. De plus, cela nous permet de spécifier le comportement des actions partagées en utilisant $\delta_{g_{super}}$. À noter que les super-groupes incluent tous les membres des autres groupes. Par conséquent, il suffit de chiffrer l'action une fois avec $K_{g_{super}}$, et elle apparaîtra dans la trace de tous les pairs impliqués. Ce type d'action est appelée « action multigroupe ».

5.3.1 IMPLÉMENTATION DE δ AVEC AUTOMATE ÉTENDU

Pour constituer une spécification prenant en compte les différentes contraintes mentionnées dans la Section 5.1.3, nous exprimons le cycle de vie en utilisant un ensemble d'automates (un pour chaque groupe). Étant donné que l'alphabet de l'automate de tous les groupes réunis Σ est un sous-ensemble d'actions et que l'alphabet de l'automate d'un groupe g est lui-même un sous-ensemble de Σ, alors $\Sigma_g \subseteq \Sigma \subseteq A$. On divise Σ_g en deux ensembles : $\Sigma_g = L_g \cup B_g$. L_g contient les actions *locales*, c.-à-d. les actions entreprises au nom du groupe g. B_g contient les

actions *frontalières*, c.-à-d. les actions effectuées par d'autres groupes mais partagées avec le groupe g (ce sont des actions multigroupes). Les actions frontalières servent à synchroniser les actions entre les groupes. Une fonction $\mathscr{S}_g : B_g \to 2^G$ assigne les actions frontalières à un ensemble de groupes qui pourraient les réaliser, à des fins de vérification.

À chaque groupe est assigné un un automate $\mathscr{A}_g = \langle Q_g, \Sigma_g, \Delta_g, q_g^0, F_g, \mathscr{S}_g \rangle$, où : Q_g est un ensemble d'états, $\Delta_g : Q_g \times \Sigma_g \to Q_g$ est la fonction de transition de l'automate, q_g^0 est son état initial, et F_g un ensemble d'états acceptants. De plus, nous ajoutons un état invalide $q^{\text{fail}} \notin F$ tel que $\forall s \in \Sigma_g : \Delta(q^{\text{fail}}, s) = q^{\text{fail}}$. Nous étendons la fonction de transition Δ_g à Δ'_g pour pouvoir vérifier la séquence de pair-actions :

$$\Delta'_g(q, \langle a', p', g', h' \rangle) \triangleq \begin{cases} q' & \text{si } \Delta_g(q, a') = q' \\ & \quad \wedge((a' \in L_g) \vee (a' \in B_g \wedge \exists g'' \in \mathscr{S}_g(a') : M(p', g''))) \\ q & \text{si } a' \notin \Sigma_g \\ q^{\text{fail}} & \text{sinon} \end{cases}$$

En commençant par un état q et étant donné une pair-action authentifiée et déchiffrée, nous vérifions les contraintes d'intégrité sur a' et vérifions le prochain état. Si l'action est frontalière, nous nous assurons que le pair à l'origine de l'action appartient à un groupe autorisé à l'exécuter par la fonction $\mathscr{S}_g$. Dans le cas où l'action n'est pas apparentée à la spécification de groupe $\mathscr{A}_g$, c.-à-d. qu'elle n'est pas présente dans l'alphabet Σ_g, nous l'ignorons simplement. Exécuter une séquence de pair-actions $(pa_0, \ldots, pa_n)$ peut donc être fait comme suit :

$$\Delta_g^+(q, (pa_0, \ldots, pa_n)) \triangleq \begin{cases} \Delta_g^+(\Delta'_g(q, pa_0), (pa_1, \ldots, pa_n)) & \text{si } n > 0, \\ \Delta'_g(q, pa_0) & \text{sinon} \end{cases}$$

Vérifier une séquence de pair-actions pour un groupe peut être fait comme suit :

$$\delta_g(s) \triangleq \begin{cases} \bot & \text{si } |s| > 0 \wedge \Delta_g^+(q_g^0, s) \notin F_g \\ \top & \text{sinon} \end{cases}$$

En exécutant la séquence de pair-actions en utilisant Δ_g^+ en partant de l'état initial (q_g^0) et en passant par des états non-acceptants, nous concluons que le cycle de vie a été violé, c.-à-d. $\delta_g(s) = \bot$.

5.3.2 GESTION DE GROUPES DYNAMIQUES

Un cycle de vie de document peut potentiellement être *grand*, c.-à-d. que les étapes que le document doit traverser sont nombreuses ou étalées dans le temps. Dans la gestion de ce type de scénario, un problème qui peut donc se poser est que les groupes puissent être modifiés alors que le document circule toujours et que son cycle de vie n'est pas achevé. Par conséquent, nous proposons une approche pour prendre en compte les changements intra-groupe. Une approche directe implique de créer un nouveau groupe à chaque fois qu'un groupe est modifié, c.-à-d., dès lors qu'un utilisateur rejoint ou quitte un groupe. Avec cette approche, le nouveau groupe hérite de la sous-spécification de l'ancien groupe, et le prédicat d'appartenance $\mathrm{M}(p,g)$ est mis à jour de manière appropriée. On associera au nouveau groupe une nouvelle clé symétrique [36]. Une nouvelle clé assure que l'utilisateur rejoignant le groupe ne peut voir les actions antérieures à son entrée, et que celui qui le quitte ne puisse plus voir les actions postérieures à son départ. Cela garantit donc que toutes les actions dans le cycle de vie restent valides après un changement d'appartenance.

36. La nouvelle clé peut être négociée en utilisant des protocoles d'échanges de clés existants (Ateniese *et al.*, 2000).

Cependant, puisque tous les groupes ne peuvent être répertoriés à l'avance, il n'est pas possible de prendre en compte les changements avant qu'ils aient lieu dans le prédicat d'appartenance. Une solution est de centraliser les informations sur l'appartenance, la rendant disponible pour les pairs de façon centralisée. Ainsi, les groupes et les membres assignés sont modifiés à un seul endroit, et les pairs peuvent simplement effectuer une requête au point de centralisation pour la vérification.

Une autre solution, plus flexible, consiste à commencer le processus avec les informations initiales concernant les groupes et leurs membres, et de les mettre à jour avec des actions spécifiques, insérées dans la séquence même. Nous proposons deux actions : `join` et `leave`, pour respectivement rejoindre un groupe et le quitter. Puisque l'appartenance à un groupe est publique, ceci nécessite qu'un groupe public $g_{\text{pub}} \in G$ existe pour entreprendre ces actions.

Bien que nous n'ayons abordé que l'aspect public de l'appartenance aux groupes, il est aussi possible de créer des groupes secrets. Dans ce cas, les changements d'appartenance ne sont pas diffusés à un groupe public, mais à tout groupe intéressé par ces changements. Cependant, ceci limite la possibilité de valider le cycle de vie par les membres extérieurs : un membre extérieur à un groupe secret ne peut pas vérifier l'appartenance du pair réalisant une action à ce groupe, il ne peut que vérifier l'appartenance des membres des groupes auxquels il appartient.

Modifier l'appartenance d'un pair p à un groupe g requiert une nouvelle action qui doit être placée dans le cycle de vie. Nous définissons deux actions : $\langle \texttt{join}, p, g, g' \rangle$ et $\langle \texttt{leave}, p, g, g' \rangle$ pour indiquer respectivement le fait de rejoindre et de quitter un groupe. Dans le cas d'un `join`, puisque initialement l'utilisateur n'est pas dans le groupe, ceci requiert qu'un autre pair p' membre de g (c.-à-d., un pair p' tel que $\mathrm{M}(p', g) \wedge p \neq p'$) soumette l'action. Lorsqu'elle est ajoutée à la séquence de pair-actions, l'action est signée avec la clé publique

du groupe $K_{g_{\text{pub}}}$, pour l'annoncer à tout le monde. À noter que, puisque la modification d'appartenance est présentée comme une action concernant une personne dans un groupe, son comportement peut être contrôlé pour être conforme avec la spécification de groupe publique (δ_{pub}). Puisque tous les pairs appartiennent à g_{pub}, l'action sera toujours présente dans la séquence déchiffrée. Ainsi, il est aussi possible de définir des politiques par groupe, permettant de définir qui peut autoriser une personne à rejoindre ou quitter un groupe. Dans le cas d'une adhésion, puisque l'action est soumise au nom de p par un autre pair p', il est aussi possible d'exiger que celle-ci soit aussi signée par p à des fins de confirmation.

Lors de la modification d'un groupe par une des deux solutions présentées ici, un nouveau groupe g' est créé avec $\delta_{g'} = \delta_g$, contenant tous les membres de g avec l'ajout ou la suppression de p.

5.4 ACCÉLÉRATION DE L'ÉVALUATION ET SUPPRESSION DES PRÉFIXES

Dans cette section, nous présentons un mécanisme permettant d'accélérer la vérification de la séquence du document. Celle-ci repose sur la réalisation d'un type spécial d'action, les actions ι, qui doivent être stockées à la fois dans la mémoire du pair et la séquence du document.

5.4.1 PAIRS SANS ÉTAT (*STATELESS*) ET AVEC ÉTAT (*STATEFUL*)

Jusque-là, les pairs impliqués dans l'échange peuvent être complètement *sans état* (*stateless*) : il n'est pas nécessaire pour eux de garder quelque information que ce soit entre leurs différents accès au document, à l'exception de leur paire de clés publique/privée. [37]

37. Ils doivent également se souvenir de la fonction de cycle de vie à appliquer ; cependant cela pourrait même être sauvegardé dans le document et chiffré avec leur clé privée. De toute façon, la fonction pourrait être la même pour tous les documents, et être directement codée en dur dans la mémoire en lecture seule du pair.

L'historique et la vérification peuvent être reconstruits à partir du document initial à tout instant.

Cependant, puisque l'historique croît au cours du temps, le temps de traitement total sur la durée de vie du document sera une fonction quadratique de la taille de l'historique : à chaque nouvelle action, la vérification devra être rejouée depuis le début. D'un autre côté, un pair *avec état* (*stateful*, l'inverse de *stateless* ; un pair qui garde donc en mémoire des informations sur l'état actuel de l'échange), peut gagner du temps sur le traitement : pour chaque document, un tel pair peut sauvegarder le condensat et l'état du document à chaque fois qu'il le reçoit. Lors d'une réception postérieure, il n'aurait qu'à vérifier le condensat et contrôler le contenu du document jusqu'au dernier instant sauvegardé localement (ceci est possible puisque la probabilité pour qu'un document falsifié ait le même n-ième condensat est extrêmement faible). De cette façon, chaque élément de la séquence de pair-actions ne nécessite qu'un seul traitement.

5.4.2 LES ACTIONS ι

Nous présentons maintenant un mécanisme permettant à un pair p de « figer » une partie de la séquence de pair-actions, de telle façon qu'il n'aura pas besoin de la reconstruire ou de la revérifier. Pour cela, nous introduisons une action spéciale idempotente, (ι_p, k) où ι est un nom quelconque et k est une donnée arbitraire. Le but de l'action ι est de permettre à un pair de réaliser une action pour « sauvegarder » dans le document un instantané (*snapshot*) de son contenu actuel, ainsi qu'une partie de l'état interne du pair qui pourra être récupérée pour reprendre la vérification du cycle de vie au point approprié. En termes de contraintes de cycle de vie, les actions ι sont simplement ignorées, puisqu'elles ne représentent aucune manipulation du document.

Techniquement, un pair p qui souhaite sauvegarder un tel instantané dans la séquence de pair-actions remplace simplement une action normale `a` par une paire (ι_p, k), et ne la chiffre pas à l'aide d'une clé de groupe. Le calcul du condensat (hachage et chiffrement avec la clé privée du pair) est effectué de manière usuelle. Les falsifications de son contenu peuvent être détectées en contrôlant le condensat, comme toute autre action.

Quand p reçoit une séquence, il peut la lire à rebours et la vérifier comme d'habitude ; cependant, cette validation peut être arrêtée dès lors qu'il rencontre une action ι_p. Si le déchiffrement de la valeur associée k retourne une chaîne de caractères jugée valide, p peut affirmer que la validation du préfixe de la trace jusqu'à ce point a déjà été faite, et récupérer le contenu associé au document à partir de k. De plus, si k contient une information à propos de l'état interne de p, il peut être utilisé pour restaurer l'état dans lequel était p à ce point précis de la séquence de pair-actions. L'évaluation de la politique de cycle de vie peut ensuite reprendre à partir de ce point jusqu'à la fin de la séquence.

De cette façon, les actions ι permettent à un pair « d'avancer » l'évaluation d'une trace au dernier « point de contrôle » enregistré dans la séquence. Ainsi, même si un pair ne sauvegarde ni son état dans le document ni son avancée dans la politique de cycle de vie entre les accès, l'évaluation n'a pas à être réalisée depuis le début à chaque fois.

Ce mécanisme peut être rendu plus puissant encore si les pairs sont autorisés à garder une copie persistante du contenu du document. Au lieu de sauvegarder le contenu du document dans une action (ι_p, k), le pair peut plutôt effectuer une action (ι'_p, k'), où k' est un nombre séquentiel chiffré avec la clé privée de p. Le but d'une telle action est d'indiquer que p a gardé un instantané *local* du document à ce stade, et qu'il a alors été jugé conforme à ses politiques de cycle de vie.

Quand un pair reçoit une séquence, il peut chercher la plus grande position i telle que, pour tout pair p, il existe une action (ι'_p, k') à une position j pour $j \geq i$. Ceci représente le dernier point dans la séquence qui a été gardé en mémoire par tous les pairs. Quand le document est retransmis à d'autres pairs, le préfixe de la séquence jusqu'à la position i peut donc être supprimé. Tenter de supprimer plus que ce préfixe peut être détecté par au moins un pair, puisque chacun garde en mémoire la dernière action (ι'_p, k') qu'il a ajouté à la trace. Recevoir une séquence qui ne contient pas cette action signifie que sa validité ne peut plus être garantie par ce pair.

5.5 APPLICATION À UN EXEMPLE MÉDICAL

Nous allons maintenant revisiter le cas d'utilisation de la Section 1.1 et illustrer comment les contraintes informelles exposées plus tôt peuvent être exprimées formellement au moyen des concepts introduits dans ce chapitre. Nous nous concentrerons sur le scénario du cas d'utilisation simplifié qui consiste au remplissage du fichier d'un patient par l'infirmière, et l'approbation par la compagnie d'assurance. Nous utilisons les pairs suivants : un docteur d, une infirmière n, la compagnie d'assurance i, et le pharmacien r. Le pharmacien est utilisé pour illustrer la notion de confidentialité dans ce scénario. Nous considérons alors les groupes suivants : $G = \{\texttt{nurse}, \texttt{ins}, \texttt{ph}, \texttt{doctor}, \texttt{nurse/ins}, \texttt{hosp}, \texttt{pub}\}$. Les quatre premiers représentent respectivement les groupes pour les infirmières, les assurances, les pharmaciens et les médecins. Pour cet exemple, chacun de ces groupes contient un seul pair, ces quatre premiers groupes contiennent donc respectivement n, i, r et d. Le groupe `nurse/ins` est un super-groupe de `nurse` et `ins`. Il est utilisé pour assurer la confidentialité de la communication du numéro d'assurance. Le groupe `hosp` inclut les infirmières et les médecins. Le groupe `pub` inclut tous les pairs : il s'agit du groupe public. Chaque pair possède une paire de clés

publique/privée $K_{u,p}$ et $K_{v,p}$ où $p \in \{d,n,i,r\}$. Chaque groupe possède également une clé de chiffrement symétrique partagée S_g, où $g \in G$.

5.5.1 SPÉCIFICATION DU CYCLE DE VIE

La Figure 5.7 montre deux automates caractérisant partiellement des cycles de vie. Il s'agit de la spécification pour les groupes `ins` et `nurse`. Pour des raisons de simplification, nous ne montrons que les états acceptants, les autres transitions mènent à q^{fail}. Nous notons $\texttt{field}(\texttt{write},\texttt{v})$, $\texttt{field}(\texttt{update},\texttt{v})$ et $\texttt{field}(\texttt{approve})$ pour indiquer respectivement que la valeur v est écrite dans le champ $field$, qu'elle remplace la valeur actuelle, ou que la valeur actuelle du champ est approuvée. Une action frontalière réalisable par un groupe $\texttt{g}_2$ et partagé avec un groupe $\texttt{g}_1$ est notée $\underset{g_1,g_2}{\texttt{action}}$. Pour simplifier la notation, dans l'automate du groupe $\mathscr{A}_{\text{g}_1}$, nous omettons le symbole g_1 des actions frontalières, puisqu'elles y font toutes référence. Par exemple, l'action $\texttt{num}(\texttt{write},\texttt{n})$ dans $\mathscr{A}_{\text{ins}}$ est une action frontalière associée au groupe `nurse` ($\mathscr{S}_{\text{ins}}(\texttt{num}(\texttt{write},\texttt{n})) = \{\texttt{nurse}\}$). Ceci indique que le numéro de sécurité sociale doit être rempli par un pair du groupe `nurse` et consultable par les membres du groupe `ins`. De plus, nous considérons une contrainte d'intégrité simple régissant le format de ce numéro d'assurance. Dans cette instance, celui-ci est un texte de 9 caractères, lettres ou chiffres. Par conséquent, toutes les actions manipulant le champ doivent le faire en respectant ces contraintes.

5.5.2 MANIPULATION DU DOCUMENT

Un document vide est d'abord créé, puis l'infirmière réalise la première action. Elle remplit le nom du patient avec l'action $a_0 = \texttt{name}(\texttt{write},\texttt{v}_0)$ qui écrit v_0 dans le champ *name*. Puisque le nom du patient est public, l'infirmière signe cette action avec $S_{\texttt{pub}}$. La

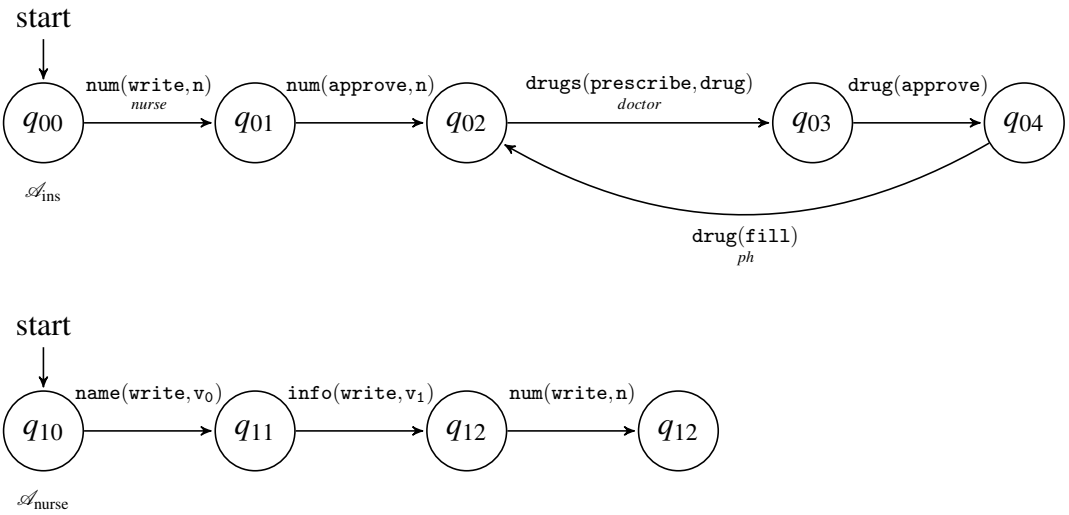


FIGURE 5.7 : Exemple de cycles de vie partiels, adapté de (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

pair-action initiale est donc $pa_0^* = \langle E[a_0, S_{\texttt{pub}}], n, \texttt{pub}, h_0 \rangle$. Le condensat h_0 est calculé par : $h_0 = E[\hbar(0 \cdot pa_0^* \cdot \texttt{pub}), K_{v,n}]$. La nouvelle pair-action chiffrée pa_0^* et les informations du groupe sont ajoutées au message vide, jointes au condensat 0, hachées, et signées par l'infirmière n en utilisant sa clé privée. Puisqu'il s'agit du premier élément de la séquence, le message précédent avait une séquence vide avec le condensat 0.

Après la saisie du nom, l'infirmière entre les données personnelles du patient en utilisant $a_1 = \texttt{info}(\texttt{write}, \texttt{v}_1)$. Étant donné que ces informations sont confidentielles et destinées au personnel de l'hôpital seulement, l'infirmière signe cette action avec $S_{\texttt{hosp}}$. La seconde pair-action est ensuite $pa_1^* = \langle E[a_1, S_{\texttt{hosp}}], n, \texttt{hosp}, h_1 \rangle$, avec $h_1 = E[\hbar(h_0 \cdot pa_0^* \cdot \texttt{hosp}), K_{v,n}]$. La troisième pair-action concerne le numéro d'assurance et est uniquement partagée confidentiellement entre l'infirmière et la compagnie d'assurance. L'action est donc $a_2 = \texttt{num}(\texttt{write}, \texttt{n})$ et chiffrée avec $S_{\texttt{nurse/ins}}$ pour former pa_2^* avec le condensat h_2. La séquence résultante est donc $\bar{s}^* = (pa_0^*, pa_1^*, pa_2^*)$.

PA	p	a	g	n	d	i	r
pa_0	n	$\texttt{name}(\texttt{write}, \texttt{v}_0)$	pub	×	×	×	×
pa_1	n	$\texttt{info}(\texttt{write}, \texttt{v}_1)$	hosp	×	×	-	-
pa_2	n	$\texttt{num}(\texttt{write}, \texttt{n})$	nurse/ins	×	-	×	-

TABLEAU 5.1 : Séquences résultantes pour chaque pair (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

5.5.3 VÉRIFICATION DE LA SÉQUENCE

Il faut maintenant contrôler et déchiffrer la séquence $\overline{s}^*$ du point de vue de la compagnie d'assurance (le pair i). Nous commençons par la dernière pair-action $pa_2 = \langle a_2^*, n, \texttt{nurse/ins}, h_2 \rangle$. Nous vérifions d'abord le condensat de la séquence en testant $\nabla^{-1}(\overline{s}^*, h_2)$. On vérifie ensuite la signature contrôlant $D[h_2, K_{n,u}] = \hbar(h_1 \cdot a_2^* \cdot \texttt{nurse/ins})$. On continue à vérifier le condensat en testant $D[h_1, K_{n,u}] = \hbar(h_0 \cdot a_1^* \cdot \texttt{hosp})$ et ainsi de suite. Il est important de noter que, alors que a_1^* n'est pas du tout connue par la compagnie d'assurance (puisqu'il s'agit des données personnelles du patient), l'intégrité de la séquence de pair-actions peut tout de même être vérifiée en validant que l'infirmière n l'a bien fait au nom du groupe $S_{\texttt{hosp}}$. Le déchiffrement de la séquence est fait en utilisant $\text{SD}(\overline{s}^*, i)$. Le pair i (l'assureur) a accès aux clés de groupe $S_{\texttt{pub}}$, $S_{\texttt{ins}}$, et $S_{\texttt{nurse/ins}}$, il est donc capable de déchiffrer a_0 et a_2. Par conséquent, la séquence pour l'assurance est $s_i = (pa_0, pa_2)$.

Les différentes séquences obtenues relativement à chaque pair sont montrées dans le Tableau 5.1. Les quatre premières colonnes détaillent les sections du tuple de la pair-action : elles listent respectivement l'identifiant de la pair-action, le pair qui a entrepris l'action, l'action elle-même, et le groupe au nom duquel a été chiffrée l'action. Les dernières colonnes représentent chaque pair. Pour chaque action, nous indiquons si la pair-action est présente (×) ou absente (-) dans la séquence du pair donné.

5.5.4 VÉRIFICATION DU CYCLE DE VIE

Une fois les séquences déchiffrées, il est possible de vérifier le respect du cycle de vie en exécutant la séquence de pair-actions sur tous les automates relatifs aux groupes auxquels appartient le pair faisant la vérification. Dans le cas de l'assurance (i) avec la séquence $s_i = (pa_0, pa_2)$, nous la vérifions sur $\mathscr{A}_{\text{ins}}$ (montré dans la Figure 5.7) en partant de q_{00}. La première action est $\texttt{name}(\texttt{write}, \texttt{v}_0)$. Cette action, cependant, n'est pas dans l'alphabet Σ_{ins}, nous l'ignorons donc et restons dans l'état q_{00}. Puisque q_{00} n'est pas un état invalide, nous continuons avec la seconde pair-action de la séquence. Il s'agit de l'action $\texttt{num}(\texttt{write},\texttt{n})$ (a_2), il y a en effet une transition vers q_{01}. De plus, puisque a_2 est une action frontalière, nous devons aussi vérifier $\mathscr{S}_i(a_2)$. Le pair auteur de l'action est l'infirmière n. L'infirmière n appartient au groupe $\texttt{nurse} \in \mathscr{S}_i(a_2)$. Par conséquent, la transition mène à q_{01}, qui n'est pas un état invalide, indiquant que le document est conforme à son cycle de vie.

L'intégrité du document peut ensuite être contrôlée pour déterminer si les contraintes d'intégrité ont été violées ou non (comme expliqué en Section 5.1.3). Le reste de $\mathscr{A}_i$ applique le scénario de la prescription de médicament. Une action $\texttt{prescribe}(\texttt{d})$ doit être réalisée par un pair du groupe `doctor`, suivie par une action locale $\texttt{approve}(\texttt{d})$ pour approuver la prescription. Un pré-requis est ajouté, à savoir que la prescription doit être remplie par un pharmacien avant que toute autre prescription soit faite. Toute séquence ne respectant pas l'ordonnancement de l'automate est rejetée et ainsi les documents non conformes peuvent être détectés.

CHAPITRE VI
IMPLÉMENTATIONS ET RÉSULTATS

Dans les chapitres précédents, nous avons présenté deux solutions techniques pour construire des historiques sécurisés d'artefact manipulés par des pairs divers et variés. La première, vue au Chapitre 4, se base sur l'adoption d'une blockchain Ethereum pour enregistrer les informations relatives à l'acheminement de colis, dans le cadre d'une simulation de logistique hyperconnectée. Ensuite, nous avons introduit au Chapitre 5 la séquence de pair-actions, une approche originale alternative aux blockchains basée essentiellement sur le chiffrement. Nous avons montré comment celle-ci pouvait être utilisée pour appliquer un exemple simplifié de cycle de vie de document médical.

Dans ce chapitre, nous détaillons les implémentations de chacune de ces méthodes et des outils associés. Nous décrivons également les expériences associées à ces approches et analysons leurs résultats ; ces expériences ont toutes été réalisées sur un ordinateur portable sous Windows 10 possédant un processeur Intel CORE i7-6700HQ cadencé à 2.6 GHz, 16 Go de mémoire vive, et un SSD de 512 Go pour le stockage ; on note également que les versions 64 bits de Java 1.8 et Go 1.13 sont installées sur le système. Les fonctionnalités offertes par chaque approche et leurs résultats expérimentaux nous permettrons finalement de conclure sur une comparaison des deux méthodes.

6.1 BLOCKCHAIN ET LOGISTIQUE HYPERCONNECTÉE

Dans le Chapitre 4, nous avons montré comment un réseau de blockchain Ethereum privé permettait la mise en place d'un système de suivi de colis. Cette implémentation s'appuie sur une simulation AnyLogic de logistique hyperconnectée et sur la définition de deux types de

smart contracts, `ShipmentManager` et `Shipment`, pour stocker les manipulations des pairs sur les colis. Nous y avons également listé un certain nombre de propriétés de cycle de vie relatives aux colis manipulés dans un contexte de logistique hyperconnectée ; ces propriétés ont été implémentées à l'aide de processeurs BeepBeep. Nous y avons également décrit une nouvelle palette BeepBeep qui permet de « transformer » les transactions de blockchain Ethereum en événements utilisables pour monitorer ces propriétés.

Afin d'évaluer l'utilisation de notre blockchain Ethereum avec cette simulation, dans cette section nous résumons les bénéfices et les manques d'une telle application dans le contexte de *supply chain*. Nous y abordons aussi ses problèmes vis-à-vis des performances, notamment en termes de vitesse et de taille de stockage. Enfin, nous analysons les résultats du monitoring des propriétés implémentées avec BeepBeep.

6.1.1 UN SYSTÈME DE SUIVI DE COLIS PARTAGÉ

Avec notre simulation reposant sur une blockchain Ethereum, plusieurs agents partagent le même réseau pour y stocker les informations des colis qu'ils manipulent, ce qui apporte plusieurs bénéfices.

Le premier concerne l'unicité des identités virtuelles des colis et l'accessibilité aux données. Dans la plupart des cas, les entreprises impliquées dans des processus de *supply chain* ne partagent pas le même système d'information. Ceci a pour conséquence des réplications superflues de données et la création de *silos* d'informations, c'est-à-dire que les données sont réparties à travers les différents acteurs et que les uns n'ont pas d'accès direct aux données des autres. Avec la blockchain, il est au contraire possible de fournir un système d'information partagé, sécurisé et fiable (qui n'est pas administré par une entité unique), de telle sorte que

l'ensemble des données est accessible par tous les acteurs à chaque instant, ce qui améliore fortement la transparence pour les entreprises et les consommateurs.

Deuxièmement, un tel système fait totalement abstraction de la nature des agents. Ceci signifie qu'il est très facile d'ajouter les agents d'une nouvelle entreprise participant au transport des colis. Il suffit alors de créer un compte pour chaque nouvel agent, puisque les transactions sont traitées de la même façon, quelque soit l'entreprise ou la nature des agents (*p. ex.*, livreurs, employés d'un *hub*). En fait, les agents n'ont même pas à être nécessairement des humains ; il pourrait en effet s'agir d'appareils IoT traitant les colis au sein des *hubs* de manière automatique. Si une entreprise ou un colis requiert des vérifications spécifiques, le système peut être adapté pour que des *smart contracts* additionnels soit déployés pour appliquer les contrôles requis.

6.1.2 TAILLE DE LA BLOCKCHAIN ET RÉPLICATION DES DONNÉES

Dans notre expérience, nous avons exécuté la simulation pendant 100 secondes, ce qui correspond à 4 heures dans la ville simulée, et observé l'évolution de la taille de la blockchain (Figure 6.1). La Figure 6.1a montre cette évolution en fonction du temps, et la Figure 6.1b en fonction du nombre d'actions. Après les 100 secondes, la blockchain atteint une taille de 532 Ko, ce qui correspond aux 318 actions envoyées par les agents AnyLogic [38]. Puisque la taille de la blockchain évolue linéairement par rapport au nombre d'actions, on peut en déduire que chaque transaction contenant une action prend environ 1.6 Ko de stockage [39].

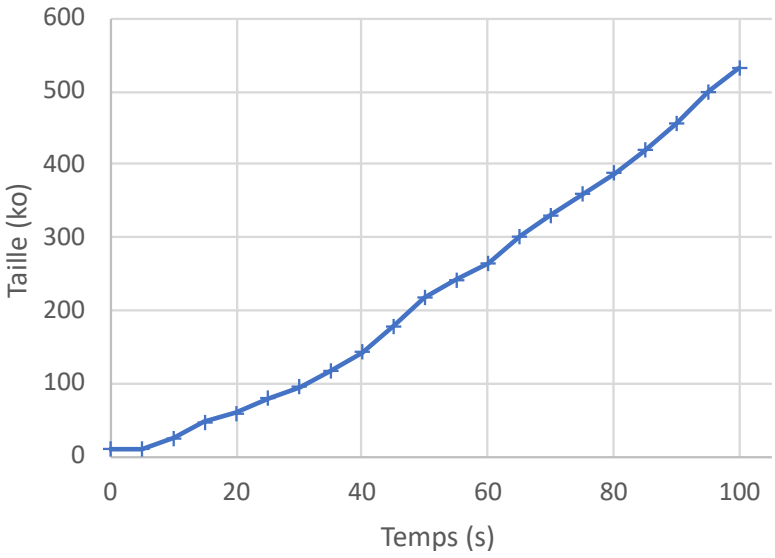
38. On peut noter qu'au début de la simulation (sans action), la taille de la blockchain est d'environ 10 Ko ; ceci correspond à l'espace requis pour la génération du bloc genesis et le déploiement du *smart contract* `ShipmentManager` par lequel les actions sont effectuées.

39. L'évolution « en escalier » à plusieurs endroits de la Figure 6.1b pourrait laisser penser que certaines actions sont plus lourdes que d'autres, mais ce résultat est en réalité dû au décalage entre l'envoi des actions par les agents et l'agrégation effective de celles-ci en blocs dans le réseau.

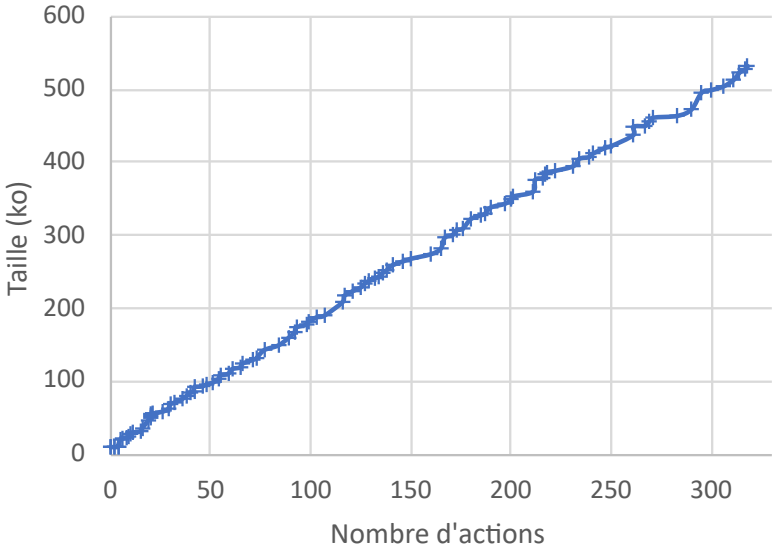
Pour de grandes entreprises de livraison comme Amazon ou UPS, ceci aurait pour résultat l'accumulation de plusieurs milliers de gigaoctets de données, potentiellement par jour. Pour de telles entités, la nécessité d'une capacité de stockage adaptée pourrait ne pas constituer un problème en soi. Cependant, il est important de rappeler que les nœuds du réseau doivent conserver la même blockchain et que la sécurité de cette dernière repose en partie sur le fait que chaque nœud en conserve une copie. Or, on s'attend à ce que le nombre de nœuds soit assez grand (*p. ex.*, dans le réseau Bitcoin on estime leur nombre à environ 100 000 (Canellis, 2019), et à 8 700 pour Ethereum (Ethernodes, 2020)), ce qui signifie qu'une quantité colossale d'informations redondantes serait conservée.

Ce problème peut être résolu en stockant les données utiles dans une base de données au lieu de les garder directement dans la blockchain. Pour assurer l'intégrité et la sécurité des données, les actions hachées sont stockées dans la blockchain, à l'instar des méthodes proposées par Liang *et al.* (2017) ou Munoz *et al.* (2019). Ainsi, les transactions possèderaient une taille fixe due au fait qu'elles ne contiennent que le condensat et non pas la totalité de l'action. De plus, si le volume de données devenait un jour trop grand, il pourrait être décidé de réinitialiser la blockchain sans que ceci supprime les données en elles-mêmes.

Cependant, cela signifie que tout un système externe doit être mis en place, géré et maintenu (en plus du réseau de blockchain), probablement par une unique entité, ce qui pourrait sembler contradictoire avec le principe même de la blockchain selon lequel aucun intermédiaire ou autorité centrale n'est nécessaire. Pour remédier à ce dernier point, il pourrait être possible d'implémenter cette base de données de manière distribuée, où chaque nœud du réseau blockchain serait également un nœud de la base de données. Ainsi, les données seraient réparties à travers l'ensemble des nœuds, avec une redondance minimale pour en assurer la sécurité, la disponibilité et la résilience, comme dans BigchainDB (BigchainDB GmbH, 2018).



(a) Taille de la blockchain en fonction du temps.



(b) Taille de la blockchain en fonction du nombre d'actions.

FIGURE 6.1 : Évolution de la taille de la blockchain : (a) en fonction du temps et (b) en fonction du nombre d'actions (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

6.1.3 CONTRÔLE D'ACCÈS ET CONFIDENTIALITÉ

Dans notre simulation, les agents connectés au réseau sont capables d'envoyer des transactions et donc de fournir les informations de suivi des colis. Chaque agent a aussi accès à l'ensemble des actions qui ont été ou seront effectuées sur les colis. Bien que ce soit un des bénéfices principaux de la blockchain, ceci peut également être une caractéristique indésirable. Un agent malveillant (ou toute personne ayant réussi à se connecter au réseau) pourrait potentiellement envoyer de fausses informations au réseau pour diverses raisons (*p. ex.*, vol, corruption de données sensibles, surcharge du réseau), et pourrait accéder à des données qu'il ne devrait pas pouvoir consulter.

Ce problème peut être résolu en déployant des *smart contracts* mettant en place divers contrôles d'accès (Zhang *et al.*, 2019) en se basant sur l'adresse de l'agent (qui est directement dérivée de sa clé publique, il n'est donc pas possible de la falsifier puisque cela ne correspondrait pas à la signature de la transaction). Ces *smart contracts* pourraient, par exemple, autoriser uniquement une liste spécifique d'adresses à ajouter certaines actions. En outre, cette liste pourrait être stockée et mise à jour directement dans la blockchain via un *smart contract*.

Cependant, cette solution protège certes contre les tentatives malveillantes d'écriture, mais ne résout pas le problème de potentiels accès en lecture. En effet, toute donnée envoyée vers un *smart contract* est conservée dans la blockchain. Même si un *smart contract* interdit l'appel d'une de ses méthodes à l'attaquant, ce dernier n'aurait qu'à parcourir l'ensemble des transactions pour reconstituer les informations cherchées. Pour remédier à ceci, deux solutions nous paraissent possibles à l'heure actuelle : 1) chiffrer les données confidentielles et distribuer les clés aux agents (ce qui nécessite la mise en place de tout un système de gestion de clés), ou 2) stocker les données dans un système externe (avec un contrôle d'accès approprié) et ne conserver que leur condensat dans la blockchain, comme nous l'avons mentionné dans la

section précédente (à noter que ce système sera alors probablement la cible des attaques et deviendra un point de défaillance unique).

6.1.4 VÉRIFICATION DES PROPRIÉTÉS AVEC BEEPBEEP

La seconde phase d'expérimentation vise à mesurer les performances des moniteurs chargés d'observer le cycle de vie de chaque colis. En effet, le principal intérêt de l'approche par blockchain n'est pas uniquement de conserver la totalité des événements dans un registre distribué, mais également d'utiliser ce dernier pour s'assurer que chacun des colis suit bien le trajet prévu.

Par conséquent, pour évaluer la scalabilité de la méthode proposée en ce qui concerne le monitoring de propriétés, nous avons exécuté chacune des chaînes de processeurs illustrées dans la Section 4.4.2. Nous utilisons pour cela des séquences simulées de 500 000 transactions issues de la blockchain, en faisant varier le nombre de colis simultanément en transit entre 10 et 10 000, et le nombre de *sauts* [40] réalisés par chaque colis entre 10 et 100. La probabilité pour qu'un colis soit dérouté lors de chaque saut sur son chemin est de $\frac{1}{5}$. Toutes les expériences et données sont disponibles par téléchargement (Betti & Hallé, 2019) sous la forme d'une instance indépendante et autonome d'environnement expérimental LabPal (Hallé *et al.*, 2018a). Le principe de LabPal consiste en ce que tout le code nécessaire, les bibliothèques et les données utilisées pour réaliser les expériences soient incorporés dans un unique fichier exécutable, de façon à ce que tous puissent télécharger et reproduire facilement les expériences d'un tiers.

Le premier élément à évaluer est la scalabilité de la vérification de propriété vis-à-vis du nombre d'événements reçus. Un exemple est montré dans la Figure 6.2 pour la Propriété 4,

40. Lorsqu'un colis passe d'un certain emplacement à un autre (c.-à-d., de son point de ramassage à un *hub*, d'un *hub* à un autre, ou d'un *hub* à sa destination finale), nous disons qu'il « saute » entre ces points. Ceci est motivé par le fait que, lors d'un déplacement d'un colis, seuls les points de départ et d'arrivée sont enregistrés dans la trace, donnant l'impression que le colis réalise un « saut ».

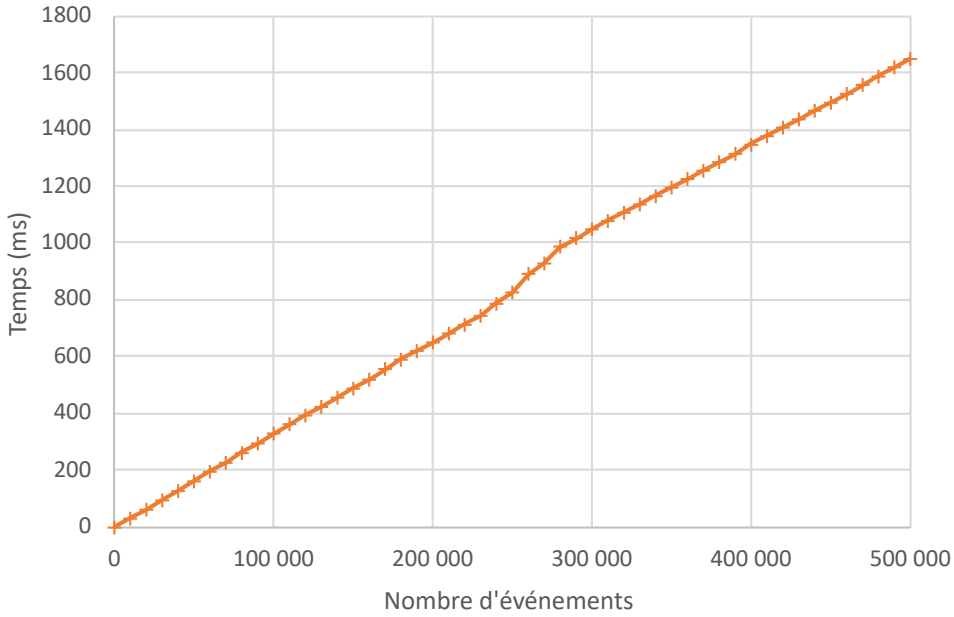


FIGURE 6.2 : Temps de traitement cumulé par rapport au nombre d'événements, pour la Propriété 4, avec 10 000 colis et 100 sauts (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

qui stipule qu'un colis doit se rapprocher de sa destination finale. On peut y observer que le temps d'exécution cumulé suit une tendance linéaire ; ceci signifie que le temps requis pour traiter un événement de la blockchain est indépendant du nombre d'événements précédemment reçus. Cette condition est essentielle pour que la vérification de respect de cycle de vie soit réalisable en pratique. Bien que ce ne soit pas illustré ici, toutes les autres propriétés exposent un comportement similaire par rapport au nombre total d'événements.

Un autre aspect digne d'intérêt est de voir comment la vérification des propriétés évolue par rapport au nombre de sauts (qui sont des événements distincts dans le cycle de vie de chaque colis), et au nombre de colis qui transitent simultanément dans le réseau. La Figure 6.3 montre le débit, en nombre d'événements consommés par seconde, de chaque propriété en fonction d'un nombre variable de sauts par colis. La Figure 6.4 montre également le débit pour

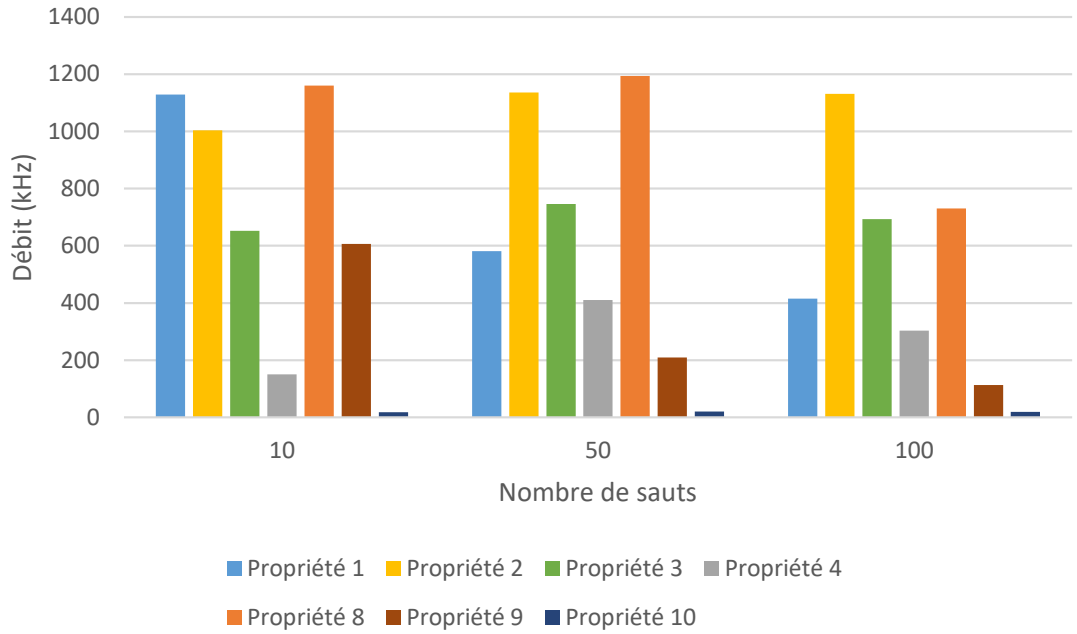


FIGURE 6.3 : Débit pour chaque propriété, en milliers d'événements pas seconde, par nombre de sauts, pour 10 000 colis (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

chaque propriété, mais cette fois-ci en fonction d'un nombre variable de colis simultanément en transit dans la simulation.

Bien que les résultats fluctuent quelque peu selon les propriétés, on peut néanmoins observer que les variations du nombre de colis et de la taille du cycle de vie des colis ne semblent pas grandement impacter la vérification de la majorité des propriétés. Ceci peut être expliqué par le fait que la plupart d'entre elles réalisent une étape de calcul à chaque événement de la blockchain ; qu'un événement appartienne au cycle de vie d'un colis ou d'un autre ne provoque aucune différence dans les coûts de calculs, si ce n'est la création d'une autre « tranche » spécifique au colis en mémoire [41]. Cependant, un nombre plus important de colis implique généralement un plus grand taux d'événements par unité de temps ; par

41. Le processeur `Slice` génère des instances de chaînes de calculs pour chaque nouveau colis rencontré, créant ainsi des « tranches ».

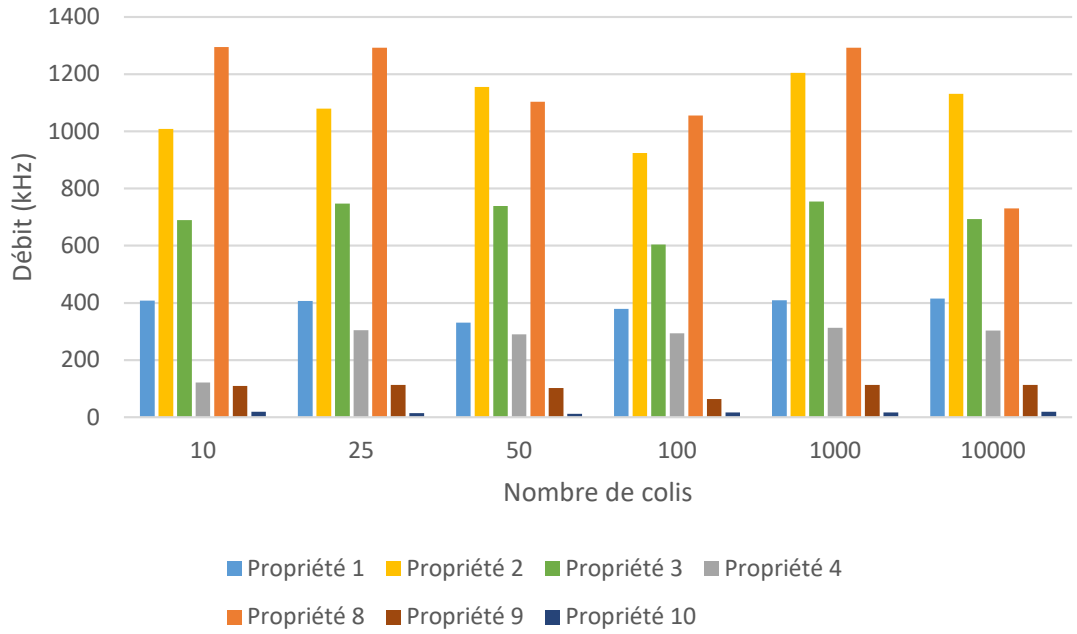


FIGURE 6.4 : Débit pour chaque propriété, en milliers d'événements par seconde, par nombre de colis, pour 100 sauts (Betti *et al.*, 2020). Reproduit avec la permission de Springer.

exemple, une simulation avec 100 colis générera probablement des événements dix fois plus vite qu'une simulation avec 10 colis.

Ainsi, une mesure plus intéressante encore est de diviser le débit global par le nombre de colis dans la simulation ; cette valeur donne une indication du nombre d'événements par seconde que chaque colis peut générer. Appliquer cette opération sur les résultats de la Figure 6.3 montre que ce « débit par colis » diminue lorsque le nombre de colis augmente. En l'occurrence, la plus lente de toutes les propriétés, la Propriété 10 (une propriété complexe qui compare le temps moyen de livraison d'une fenêtre de colis par rapport à une ancienne fenêtre), peut traiter près de 18 000 événements par seconde pour 10 000 colis. Ceci signifie que chaque colis pourrait générer environ 1,8 événements par seconde dans la blockchain et que le moniteur pourrait tout de même réaliser la vérification des propriétés en temps réel.

6.1.5 PERFORMANCES

Il est cependant nécessaire de contraster ces résultats avec les performances générales de la blockchain Ethereum. Dans notre expérience, nous avons pu voir que l'exécution de la simulation avait généré 318 événements en 100 secondes, ce qui correspond donc à 318 événements en 4 heures simulées, soit 0,022 événement par seconde. Bien qu'elle représente une application concrète pertinente de la blockchain à la logistique hyperconnectée, la simulation présentée n'est cependant pas du tout représentative de la réalité en termes d'échelle.

En effet, lors de mon stage de recherche au *Physical Internet Center* (laboratoire dirigé par Benoit Montreuil, faisant partie du département de *supply chain* et logistique au *Georgia Institute of Technology*), un projet en particulier mené par les équipes du laboratoire concernait la mise en place d'une infrastructure de logistique hyperconnectée dans la ville de Shenzhen, en Chine. Dans cette infrastructure, on s'attend à ce qu'environ un million de colis transitent à travers le réseau dans une journée ordinaire, chaque colis suivant une route contenant entre 2 et 8 sauts ; ceci générerait un total d'environ 80 millions d'événements par jour. Dans les jours de forte demande, ce volume peut être multiplié par cinq, ce qui donne 400 millions d'événements par jour dans le pire des scénarios. Le traitement de ces événements en temps réel requiert alors un débit de 4 600 Hz. Les figures 6.3 et 6.4 montrent que les capacités calculatoires d'un unique ordinateur modeste excède cette valeur ; même le plus lent des moniteurs (celui de la Propriété 10) a un débit d'environ 12 000 Hz, ce qui suffirait à traiter près de trois fois le pic d'activité estimé.

Si les moniteurs ne présentent aucune difficulté à traiter cette quantité d'événements, on ne peut pas en dire autant des réseaux blockchain. En effet, nous avons vu en Section 3.4.4 que le meilleur des réseaux actuels (EOS) peinait à atteindre les 4 000 transactions par seconde

(TPS), ce qui est en deçà des capacités de traitement requises pour ce scénario. De plus, il est important de préciser que ce dernier ne concerne qu'une seule entreprise de livraison, dans une unique ville. Si l'on souhaite mutualiser les réseaux de plusieurs villes d'un même pays, voire du globe, en incluant les nombreux acteurs que cela implique, n'importe quel réseau de blockchain (même la promesse de 300 000 TPS de Futurepia) serait probablement complètement dépassé. En outre, comme nous l'avons mentionné plus tôt, nous ne prenons en compte ici que les événements correspondant à des ramassages et livraisons de colis ; en situation réelle, d'autres éléments comme la gestion interne des colis dans les *hubs*, ou la collecte d'informations périodiques spécifiques à certains colis (*p. ex.*, le suivi de la température d'aliments tout au long de leur transport pour assurer le respect de la chaîne du froid) seraient également à considérer.

6.2 IMPLÉMENTATION DES SÉQUENCES DE PAIR-ACTIONS : LA BIBLIOTHÈQUE ARTICHOKE-X

Dans le Chapitre 5, nous avons introduit les séquences de pair-actions, une approche ayant pour but de remédier à certains inconvénients de la blockchain. Nous y avons vu que nous étions capables de construire des historiques d'artefacts à la fois confidentiels, authentifiés et immuables, permettant une vérification du cycle de vie en termes d'ordre des actions, d'intégrité des informations, et de contrôle d'accès.

Notre première implémentation des séquences de pair-actions est composée de deux outils distincts. Le premier est Artichoke-X, une bibliothèque Java disponible gratuitement sous licence open source [42]. Artichoke-X utilise les fonctions cryptographiques incorporées dans Java (telles que RSA et MD5) ; pour manipuler les méta-données de fichiers, nous nous

42. https://github.com/liflab/artichoke-x

appuyons sur Apache Tika [43], une bibliothèque polyvalente qui peut lire et écrire des méta-données dans plusieurs centaines de formats de fichier. Ceci inclut les types de document couramment utilisés telles que les documents Microsoft Office, les fichiers PDF, les fichiers images et audio, et même le code source.

6.2.1 UTILISATION

La bibliothèque est utilisée en manipulant par programmation des objets de haut niveau tels que les pairs, les actions et les groupes. La première étape consiste à créer (ou récupérer) les instances des objets pour un scénario spécifique. Dans l'extrait de code de la Figure 6.5, nous créons un pair et un groupe, et associons une paire de clés publique/privée RSA à chacun. Une instance d'objet `Action` est aussi créée. Dans cet exemple, une action est simplement représentée par une chaîne de caractères.

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");

Peer alice = new Peer("Alice", Cipher.getInstance("RSA"));
alice.setKeyPair(generator.generateKeyPair());

Group g1 = new Group("Group 1", Cipher.getInstance("RSA"));
g1.setKeyPair(generator.generateKeyPair());

Action a = new Action("a");
```

FIGURE 6.5 : Initialisation d'un pair, d'un groupe, de leurs clés, et d'une action avec Artichoke-X (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

La prochaine étape est de créer une séquence de pair-actions vide, et de commencer à y insérer des actions (Figure 6.6). Ceci est réalisé en créant un `HistoryManager`, qui est

43. https://tika.apache.org

responsable de la manipulation de l'objet `History`. Le `manager` est d'abord configuré avec les ensembles de pairs et de groupes possibles, ainsi qu'avec les actions pouvant être rencontrées. Dans l'extrait ci-dessous, un nouveau manager est créé et chargé d'utiliser le MD5 comme fonction de hachage. Une action `a`, effectuée par Alice au nom du groupe `g1`, est ensuite ajoutée à la séquence.

```
HistoryManager manager = new HistoryManager(MessageDigest.getInstance("MD5"));
manager.add(alice, bob, ...);
manager.add(g1, g2, ...);
manager.add(a, b, ...);

History h = new History();
manager.appendAction(h, alice, a, g1);
```

FIGURE 6.6 : Ajout d'une action à l'historique avec Artichoke-X (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

Le manager peut être aussi chargé de vérifier une séquence de pair-actions existante (Figure 6.7). L'appel de la méthode `isHistoryValid` envoie une exception détaillant les raisons de la violation, et indique en particulier à quelle position l'élément frauduleux de la séquence de pair-actions est situé.

La dernière étape du processus consiste à évaluer le respect du cycle de vie sur une séquence de pair-actions. Ceci est fait par l'implémentation de l'interface `Policy`. Cette interface définit deux méthodes : la première, `evaluate`, prend en argument un unique élément de l'historique (consistant en un pair, une action et un groupe). Cette méthode est censée être appelée successivement sur chaque élément de la séquence de pair-actions, et soulève une exception dès lors que l'on considère la politique de cycle de vie violée. La seconde méthode, `reset`, indique simplement que le processus d'évaluation doit être recommencé à

```
try {
  manager.isHistoryValid(h);
}
catch (InvalidHistoryException e) {
// La sequence est invalide
}
```

FIGURE 6.7 : Vérification d'une séquence de pair-actions avec Artichoke-X (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

zéro. Par exemple, l'extrait de code de la Figure 6.8 est une politique qui vérifie qu'Alice ne peut exécuter l'action `a` plus d'une fois sur le document.

Étant donné un objet `Policy`, le manager peut être chargé d'évaluer le respect de cette politique pour une séquence de pair-actions donnée, comme c'est le cas dans la Figure 6.9.

Comme on peut le voir, la bibliothèque Artichoke-X est très flexible dans beaucoup d'aspects. D'abord, elle permet l'utilisation de fonctions arbitraires pour le hachage et le chiffrement asymétrique. Deuxièmement, les actions peuvent être définies comme on le souhaite, tant qu'elles peuvent être sérialisées en une chaîne de caractères. Troisièmement, l'expression d'un cycle de vie n'est pas restreinte à un langage formel en particulier (tel que les automates, la logique temporelle ou le BEDL) : le dernier exemple montre que l'implémentation d'un objet `Policy` peut contenir du code Java arbitraire, et donc peut donc techniquement accommoder n'importe quelle propriété. Cependant, si les utilisateurs souhaitent exprimer des politiques d'un plus haut niveau d'abstraction, il existe plusieurs bibliothèques (telles que SealTest (Hallé & Khoury (2017) ou BeepBeep) qui permettent d'écrire des spécifications en utilisant UML ou LTL, et de les incorporer dans des objets `Policy` d'Artichoke-X.

```
class AlicePolicy implements Policy {

  boolean seen_a;

  public void evaluate(Peer p, Action a, Group g)
         throws PolicyViolationException {
    if (p.getName() == "Alice" && a.getName() == "a")
      if (seen_a)
        throw new PolicyViolationException("Alice a execute 'a' deux fois");
      seen_a = true;
  }

  public void reset() {
    seen_a = false;
  }
}
```

FIGURE 6.8 : Implémentation de l'interface `Policy` (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

6.2.2 INTERFACE EN LIGNE DE COMMANDE

Nous avons ensuite développé en PHP Artichoke-PDF, un outil en ligne de commande pour le cas spécifique où les artefacts sont des formulaires PDF dynamiques. Dans ce contexte, les divers champs du fichier constituent les données du document, et peuvent être remplis et modifiés par plusieurs pairs. Un champ spécial caché est inclus dans le document et vise à contenir la séquence de pair-actions désignant l'historique des modifications [44].

44. À noter que ce champ est rendu invisible uniquement pour des raisons de lisibilité ; en aucun cas il ne s'agit de le protéger de potentielles falsifications.

```
Policy alice_pol = new AlicePolicy();
try {
  manager.evaluate(h, alice_pol);
}
catch (PolicyViolationException e) {
  // Policy is violated
}
```

FIGURE 6.9 : Vérification d'une politique de cycle de vie (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

Artichoke-PDF utilise LaTeX pour générer des formulaires avec des champs divers et une séquence de pair-actions vide. Il utilise aussi pdftk [45] pour extraire et manipuler les données du formulaire en arrière-plan. Bien qu'Artichoke-PDF ait été réalisé comme preuve de concept avec un minimum d'ergonomie pour l'utilisateur, il est totalement fonctionnel et son code source est publiquement disponible sous la licence GNU GPL [46]. L'implémentation actuelle supporte une version légèrement simplifiée des séquences de pair-actions, où un seul groupe existe, mais où les pairs de chaque groupe ont leur propre paire de clés publique/privée pour signer leurs actions.

Actuellement, Artichoke-PDF supporte trois actions principales réalisables sur les documents, à savoir le remplissage, l'examen et le contrôle de la séquence de pair-actions d'un formulaire. Remplir le formulaire consiste en l'écriture (ou le remplacement) d'un ou plusieurs champs avec des valeurs spécifiques, Dans ce contexte, remplir un formulaire met également à jour la séquence de pair-actions contenue dans ce formulaire pour y inclure l'action et les informations relatives au pair. Ceci est fait par la ligne de commande suivante. Par exemple, si

45. http://pdftk.org
46. https://github.com/liflab/artichoke-pdf

Alice souhaite écrire « `foo` » dans le champ `F1` du fichier `Form.pdf`, la commande à écrire est :

```
$ artichoke Form.pdf fill \
  -k private_key_Alice.pem \
  -p Alice \
  -o Form-filled.pdf \
  F1 foo
```

Ici, l'argument `-p` indique le nom du pair, `-k` la clé privée à utiliser pour calculer le condensat, et `-o` le nom du fichier PDF modifié.

Examiner le contenu d'un formulaire peut être fait par la ligne de commande suivante :

```
$ artichoke Form.pdf dump
```

Cela affichera les valeurs actuelles de tous les champs du formulaire, ainsi qu'un résumé de la séquence de pair-actions contenue dans le document, qui ressemblera à ce qui est illustré dans la Figure 6.10.

```
Form fields
-----------
F1:    baz
F2:    bar

Peer-action sequence
--------------------
Alice  W|F1|foo      Rm/MRSzK...oYpROgO=
Bob    W|F2|bar      kEvrkC+e...bX4NO1w=
Carl   W|F1|baz      F3UYg+n1.../YPs3/k=
```

FIGURE 6.10 : Contenu du fichier PDF modifié (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

La séquence de pair-actions montre qu'Alice a d'abord écrit « `foo` » dans le champ `F1`, puis que Bob a écrit « `bar` » dans le champ `F2`, et enfin que Carl a modifié `F1` avec la valeur « `baz` ». La colonne à droite est une version réduite du condensat pour chaque événement.

La dernière fonction pouvant être réalisée avec Artichoke-PDF est de valider le contenu d'un formulaire. Cela est fait de la manière suivante :

```
$ artichoke Form.pdf check key1 [key2 [...]]
```

Ceci vérifiera la séquence de pair-actions dans `Form.pdf`, en utilisant les fichiers contenant les clés publiques `key1`, `key2`, etc. si nécessaires. Cette liste de fichiers locaux agit telle une forme primitive de « porte-clés ». Bien entendu, une version plus mature de l'outil pourrait remplacer ces fichiers par le système de porte-clés de la machine de l'utilisateur, ou même envoyer des requêtes à des serveurs distants stockant des certificats de clé publiques X.509 (Cooper *et al.*, 2008).

Cette opération réalisera les trois étapes de vérification mentionnées plus tôt, à savoir : 1) assurer que le condensat de chaque événement de la séquence de pair-actions correspond à l'action et au nom du pair spécifiés ; 2) assurer que les valeurs de chaque champ du formulaire correspondent au résultat de l'application de la séquence de pair-actions sur le document vide ; 3) assurer que la séquence de pair-actions est conforme au cycle de vie.

La politique est actuellement spécifiée par du code PHP défini localement, en implémentant une fonction appelée `check_policy` qui reçoit en entrée la séquence de pair-actions du document actuel. Par conséquent, comme pour la bibliothèque Java, l'application de la politique n'est pas liée à un langage de spécification particulier, à partir du moment où il peut être exprimé à l'aide du contenu de la séquence de pair-actions uniquement.

6.2.3 PERFORMANCES

Dans cette section, nous procédons aux tests visant à mesurer la consommation en ressources de notre solution intermédiaire. En particulier, nous voulons déterminer si l'application répétée de chiffrement et de hachage induit un coût raisonnable, en termes de temps et d'espace, tandis que l'historique d'un document grandit au cours du temps. Les expériences et données sont une nouvelle fois incluses dans une instance de laboratoire LabPal disponible pour téléchargement [47].

Dans ces expériences, des paires de clés RSA de 2048 bits sont générées pour chaque pair et chaque groupe. En effet, pour simplifier le développement de cette première version, nous utilisons un mécanisme de chiffrement asymétrique pour les groupes au lieu d'un chiffrement symétrique, contrairement à la définition donnée au Chapitre 5 ; ceci est corrigé dans la Section 6.3. De plus, puisque la nature même des fichiers et leur accès en écriture ou lecture ne sont pas pertinents pour notre étude, les expériences furent réalisées directement en manipulant les séquences de pair-actions avec la bibliothèque Artichoke-X.

Nous avons tout d'abord généré une séquence de pair-actions vide, et ajouté de façon répétitive des actions arbitraires au nom d'utilisateurs et de groupes aléatoires, ce qui a pour effet de créer une séquence de pair-actions de taille croissante. En l'occurrence, les actions sont composées d'un seul octet, mais leur taille a en fait peu d'importance dans les expériences présentées (sauf éventuellement pour le temps d'exécution du chiffrement), car le chiffrement avec des clés RSA de 2048 bits résulte nécessairement en un cryptogramme de 256 octets, quelle que soit la taille du message à chiffrer [48] (Jonsson & Kaliski, 2003) ; une étude plus

47. https://datahub.io/dataset/artichoke-x-lab

48. Il faut néanmoins que celui-ci ait une taille inférieure à celle des clés.

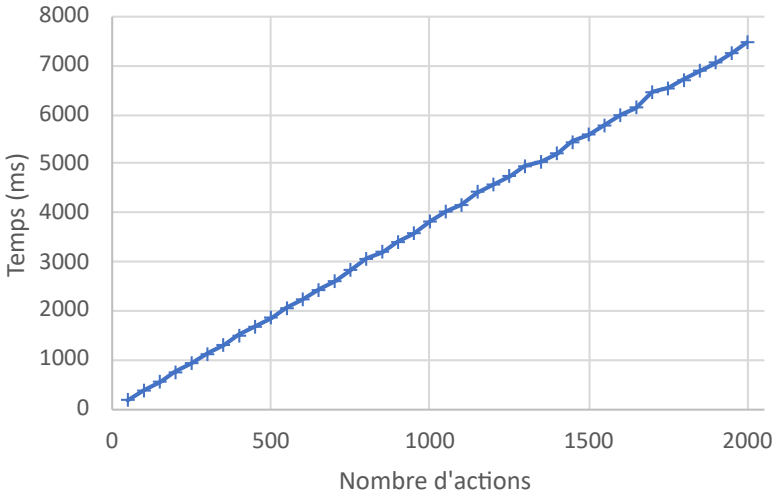
étendue sur l'impact de la taille des actions est néanmoins réalisée en Section 6.3 sur une autre implémentation des séquences de pair-actions.

Le premier facteur mesuré est le temps d'exécution pour l'ajout de nouvelles actions à une séquence existante. Ceci est illustré dans la Figure 6.11a. On peut y voir que le temps d'exécution augmente linéairement avec le nombre d'opérations en écriture. Nous pouvons déduire de ce graphique qu'il faut approximativement 3,7 millisecondes pour ajouter une seule action.

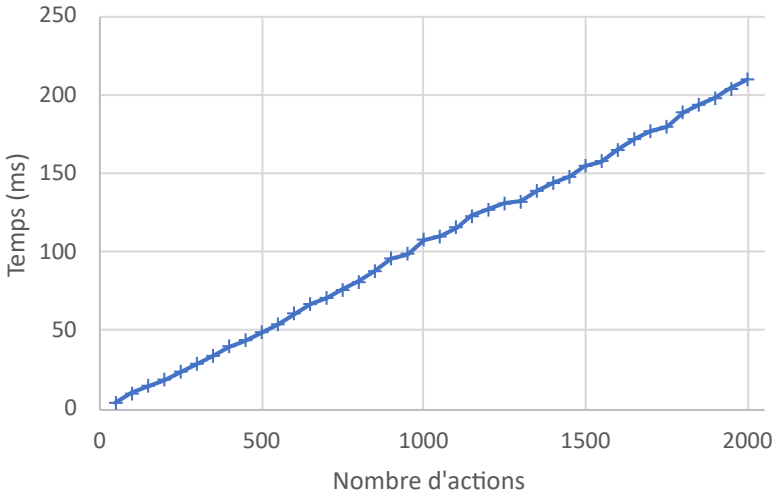
Le second facteur mesuré est le temps requis pour simplement vérifier une séquence existante sans la modifier ; cela est montré dans la Figure 6.11b. De nouveau, le temps de vérification est linéaire par rapport à la taille de la séquence, avec un temps de vérification par élément d'environ 0,1 milliseconde. On peut donc voir que lire/déchiffrer des opérations est beaucoup plus rapide que de les écrire/chiffrer.

Cependant, la Section 5.4 a introduit le concept de pair *stateful*, qui permet à un pair de réserver un petit espace de mémoire persistante pour sauvegarder le dernier élément de la séquence de pair-actions qu'il a traité et sa position i dans la séquence. Quand une séquence de pair-actions prolongeant une séquence précédente est reçue, un tel pair peut simplement vérifier que le i-ème élément de la séquence est le même que celui gardé en mémoire, et il ne lui reste alors qu'à vérifier le reste de la séquence qui a été ajouté après cette position. Un pair *stateless* lui, au contraire, n'a pas de mémoire de la séquence et doit donc la revalider depuis le début lorsqu'il en reçoit une nouvelle.

Par conséquent, l'autre expérience réalisée consiste à comparer le temps de traitement total requis pour vérifier une séquence de pair-actions d'une taille donnée entre un pair *stateful*



(a) Ajout d'actions à la séquence.



(b) Vérification de la séquence.

FIGURE 6.11 : Temps d'exécution d'Artichoke-X avec une séquence de pair-actions de taille croissante : (a) pour ajouter des actions dans la séquence et (b) pour vérifier la séquence (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

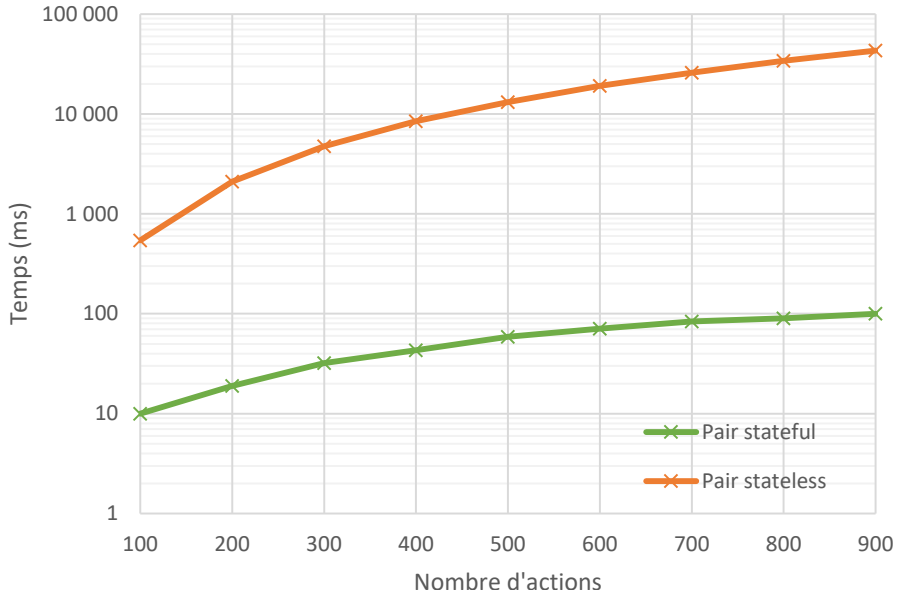


FIGURE 6.12 : Comparaison du temps de traitement total entre un pair *stateful* et un pair *stateless* pour vérifier une même séquence de pair-actions (échelle logarithmique sur les ordonnées) (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

et un pair *stateless* [49]. Les résultats sont rapportés dans la Figure 6.12. Comme on s'y attendait, l'utilisation d'un pair *stateful* améliore considérablement le temps requis pour traiter une séquence. Tandis qu'il faut seulement 60 millisecondes pour le traitement total d'une séquence de taille 500 avec un pair *stateful*, les mêmes opérations prennent 13,2 secondes avec un pair *stateless*, soit 220 fois plus de temps. Et la différence entre les deux méthodes croît avec la taille de la séquence.

Le dernier facteur mesuré est l'espace en mémoire de la séquence pour des tailles croissantes de séquences de pair-actions ; cela est montré dans la Figure 6.13. Comme on pouvait

49. Le temps de traitement total de vérification est le temps cumulé des diverses vérifications qui ont lieu lors de la construction d'une séquence de pair-actions : un pair p_a ajoute la première action, le pair p_b vérifie la séquence de pair-actions et ajoute une autre action, le pair p_a vérifie à son tour la séquence de pair-actions alors composée de deux actions et ajoute une action, et ainsi de suite. Le temps de vérification total d'une séquence de pair-actions est donc la somme des temps de ces vérifications intermédiaires, et n'est pas à confondre avec le temps de vérification « standard » (tel qu'illustré dans la Figure 6.11b).

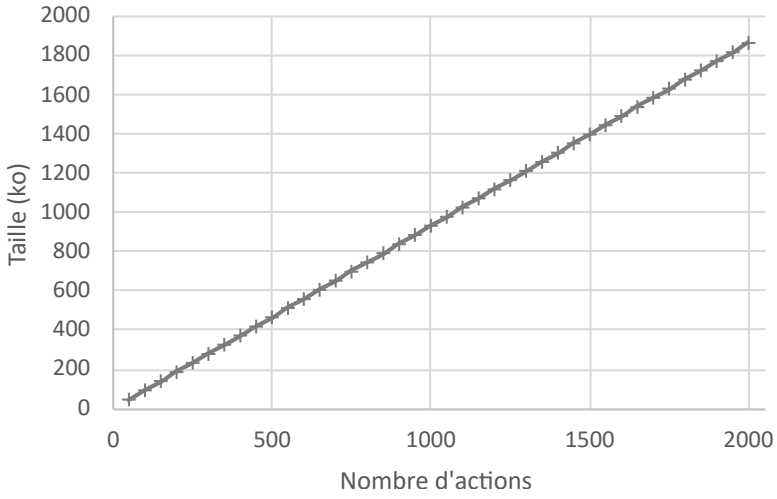


FIGURE 6.13 : Mémoire nécessaire pour une séquence de pair-actions de taille croissante (Hallé *et al.*, 2018b). Reproduit avec la permission d'Elsevier.

le prévoir, l'espace requis par la séquence croît linéairement avec le nombre d'opérations appliquées au document, indiquant que chaque élément de la séquence requiert un espace constant. Dans cette implémentation d'Artichoke-X, cet espace est d'environ 930 octets par action. À noter cependant que la forme par défaut de sérialisation dans la bibliothèque est particulièrement inefficace, puis qu'elle utilise deux applications d'un encodage en base 64, pour convertir des valeurs binaires en chaînes de caractères. De plus, puisque nous signons le condensat avec une clé RSA de 2048 bits, celui-ci occupera nécessairement 256 octets, soit 344 octets après le premier encodage en base 64, et environ 458 octets [50] après l'encodage base 64 de toute la séquence de pair-actions. Ceci signifie que sur les 930 octets nécessaires au stockage d'une pair-action, près de la moitié est occupée par le condensat seul. Sans compter que même si l'action à ajouter n'est composée que d'un seul octet, son cryptogramme par RSA sera également de 256 octets ce qui, après les deux encodages, représente là aussi près

50. La quantité exacte d'octets nécessaire dépend également du reste de la séquence de pair-actions, puisque l'encodage en base 64 peut nécessiter un remplissage (*padding*) de 1 à 3 octets si la taille de la donnée à encoder n'est pas divisible par 3 (Josefsson, 2006).

de la moitié de la taille finale de la pair-action, indépendamment du fait que l'action soit de 1 ou 256 octets [51].

Pour réduire cette consommation d'espace, des mécanismes de chiffrement et de signature plus compactes (tels que AES et ECDSA, respectivement) peuvent être utilisés pour que les tailles des valeurs binaires (et donc de leur encodage) soient optimisées. Néanmoins, même sous cette forme, on peut dire que l'encodage des séquences de pair-actions dans un document entraîne une surcharge raisonnable : une séquence de pair-actions de 1 Mo peut contenir plus de 1 075 modifications distinctes.

6.3 AMÉLIORATION DE L'IMPLÉMENTATION ET AUTRES OUTILS

Afin que les performances de l'implémentation des séquences de pair-actions soit davantage comparable avec celle de la blockchain Ethereum, nous avons décidé de développer une autre implémentation des séquences de pair-actions s'appuyant en partie sur les technologies utilisées par les clients Ethereum. Cette implémentation, appelée Go-Artichoke et disponible sous licence open source en ligne [52], a été écrite en Go, qui est le langage utilisé dans le développement de `go-ethereum` (l'implémentation du protocole Ethereum qui a été utilisée dans l'expérimentation du Chapitre 4).

Comme nous le verrons, Go-Artichoke reprend les fonctionnalités d'Artichoke-X [53] tout en se différenciant sur certains choix technologiques. Toutes ces modifications sont réalisées dans le but d'améliorer les performances et de se rapprocher, toutes proportions gardées, du niveau d'optimisation des clients Ethereum. Ainsi, nous espérons que la comparaison

51. La taille maximale du message en entrée d'un chiffrement RSA avec des clés de 2048 bits

52. https://github.com/qbetti/go-artichoke-lab

53. Elle ne reprend cependant pas les fonctionnalités liées à l'expression et l'évaluation de propriétés puisqu'elles ne concernent pas les facteurs d'optimisation mesurés ici.

Élément modifié	Go-Artichoke	Artichoke-X
Langage de développement	Go	Java
Chiffrement des actions	Symétrique (AES-GCM)	Asymétrique (RSA)
Signature des condensats	ECDSA (secp256k1)	RSA
Hachage des condensats	Keccak-256	MD5
Identification des pairs	Par leur clé publique	Par leur nom

TABLEAU 6.1 : Résumé des changements entre Go-Artichoke et Artichoke-X
© Quentin Betti.

des approches par blockchain et par séquence de pair-actions puisse être la plus pertinente possible.

Enfin, nous introduisons un outil supplémentaire, à savoir l'implémentation d'un système de porte-clés pour stocker les clés personnelles et de groupe d'un pair dans un fichier sécurisé.

6.3.1 DIFFÉRENCES ENTRE GO-ARTICHOKE ET ARTICHOKE-X

Mis à part le langage de développement, Go-Artichoke diffère d'Artichoke-X par plusieurs choix de conception et d'algorithmes. L'ensemble des différences entre ces deux implémentations est résumé dans le Tableau 6.1.

La première est l'utilisation de chiffrement symétrique pour les actions. Ainsi, conformément à la formalisation des séquences de pair-actions du Chapitre 5, chaque groupe possède une clé AES de 256 bits. Lors de l'ajout d'une action, un pair appartenant au groupe utilise alors cette clé pour chiffrer l'action avec AES. Dans ce contexte, l'un des intérêts d'AES par rapport à RSA réside dans son mécanisme de chiffrement par bloc, facilitant le chiffrement de données de taille arbitraire (là où RSA est limité par la taille de sa clé) et offrant des performances bien supérieures sur les opérations de chiffrement et déchiffrement (Dai, 2009).

En outre, les clés AES sont bien plus petites que RSA, tout en offrant un niveau de sécurité supérieur : d'après Barker (2016), l'utilisation d'AES avec des clés de 256 bits présente le même niveau de sécurité que celle de RSA avec des clés de 15 360 bits.

Il est important de noter que nous utilisons un mode spécial d'AES : le Galois/Counter Mode, ou GCM (Dworkin, 2007). Contrairement au AES classique, AES-GCM utilise des *vecteurs d'initialisation* (IV) qui permettent de rendre les données à chiffrer plus « aléatoires » et donc d'empêcher un potentiel attaquant de déduire des informations sensibles de l'observation et de la comparaison des cryptogrammes. De plus, la principale particularité d'AES-GCM réside dans la génération d'une étiquette d'authentification (*authentication tag*, ou AT) pour chaque opération de chiffrement afin d'assurer l'authenticité du cryptogramme : si un attaquant décidait de modifier les bits du cryptogramme, la vérification systématique de l'AT lors du déchiffrement permettrait de s'en rendre compte. Ceci nous permet d'avoir une couche de sécurité supplémentaire et de pouvoir distinguer deux types d'attaques : d'un côté l'AT permet de savoir si le cryptogramme a été falsifié ou si une clé de groupe différente de celle annoncée a été utilisée pour chiffrer une action, et de l'autre le condensat permet d'identifier les tentatives d'usurpation d'identité et d'altération des pair-actions dans leur ensemble. Dans notre implémentation, nous utilisons des IV et des AT de respectivement 96 et 128 bits, qui sont directement concaténés au cryptogramme de l'action dans la séquence de pair-actions.

Deuxièmement, pour signer le condensat d'une pair-action, nous décidons d'opter pour l'algorithme ECDSA plutôt que RSA. Plus particulièrement, nous adoptons la même implémentation que celle de Go-Ethereum, qui utilise la courbe elliptique `secp256k1` (Brown, 2010), très populaire dans les réseaux blockchain, avec des clés de 256 bits. Ce choix est particulièrement motivé par les clés générées qui, là encore, présentent des niveaux de sécurité supérieurs pour des tailles plus petites : des clés ECC de 256 à 383 bits offrent le même niveau de sécurité que des clés RSA de 3072 bits. Ce gain d'espace s'accompagne néanmoins d'un

inconvénient. Si la génération de signatures avec ECDSA est en général plus rapide qu'avec RSA, leur vérification, elle, est plus lente (Dai, 2009). Nous verrons cependant que ceci n'impacte que modérément le traitement des séquences de pair-actions, tout en augmentant grandement leur sécurité.

La troisième modification concerne l'adoption de Keccak-256 à la place de MD5 pour le hachage nécessaire au calcul du condensat des pair-actions. Comme nous l'avons dit dans la Section 3.1.1, MD5 est une fonction de hachage dont l'utilisation est déconseillée, notamment si elle est combinée à la génération de signatures. L'algorithme de hachage Keccak-256 (équivalent du SHA3-256), certes, est un plus lent que MD5 (Bertoni *et al.*, 2020) et produit des valeurs de hachage deux fois plus grandes (256 bits au lieu des 128 bits de MD5), mais il fournit un bien plus haut niveau de sécurité et son utilisation est recommandée par le NIST (Barker, 2016).

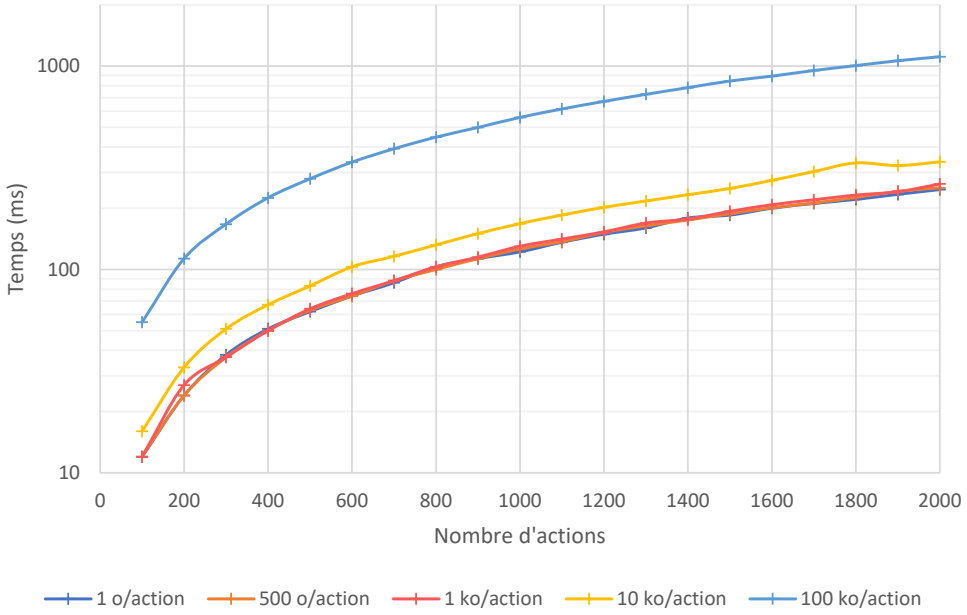
Enfin, la dernière modification réside dans l'utilisation de la clé publique d'un pair à la place de son nom pour l'identifier dans la séquence de pair-actions. De cette façon, à l'instar de la génération d'adresse de comptes Ethereum, la clé publique (encodée en hexadécimal dans la pair-action) peut ainsi servir d'identifiant universel pour le pair. De plus, cela permet aux pairs de vérifier une séquence sans avoir à connaître les clés publiques des autres pairs à l'avance. Ainsi, la vérification de séquences de pair-actions devient un processus autonome, puisque tout utilisateur peut vérifier une séquence en utilisant uniquement les informations qu'elle contient [54].

54. Cette implémentation ne prend donc pas en compte les changements de clés des pairs, car une clé obsolète ne pourrait plus servir d'identifiant unique. Il faudrait alors un système externe faisant le lien entre les pairs et leurs clés publiques, qu'elles soient obsolètes ou actuelles.

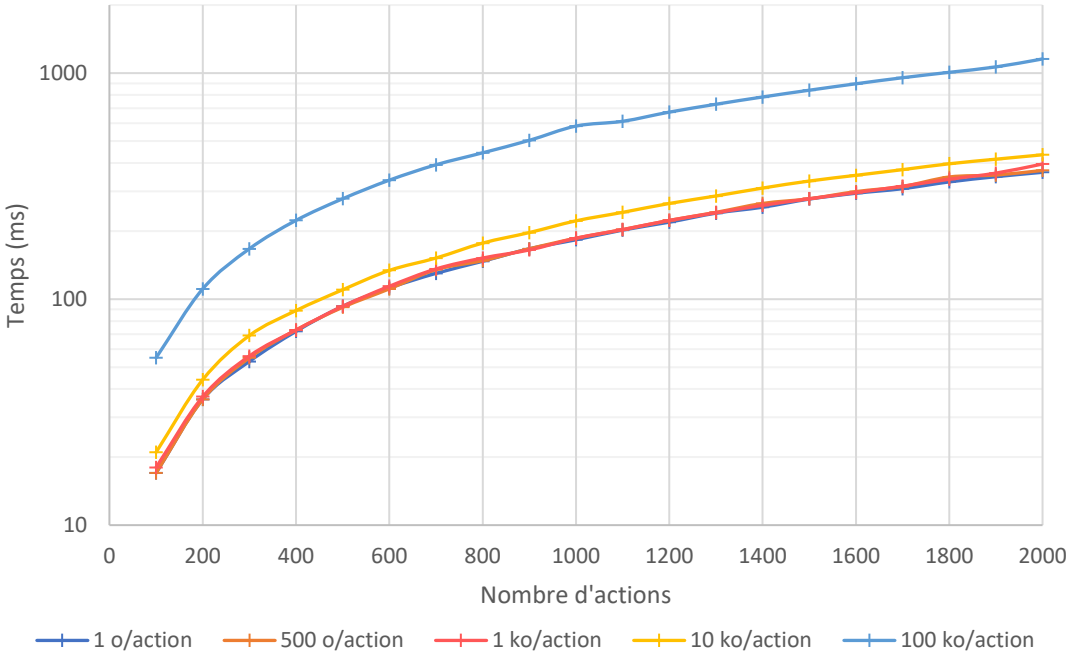
6.3.2 PERFORMANCES

Les expériences réalisées avec Go-Artichoke sont les mêmes qu'avec Artichoke-X, à l'exception de l'absence de comparaison entre pairs *stateful* et *stateless*. Cependant, cette fois-ci nous prenons en compte l'évolution des différents critères selon plusieurs tailles d'action, contrairement à la Section 6.2.3 où toutes les actions étaient composées d'un seul octet. Ainsi la Figure 6.14 présente les temps d'exécution nécessaires à l'ajout d'actions à une séquence et à la vérification d'une séquence de taille croissante, en différenciant les séquences par la taille de leurs actions. Pour la Figure 6.14a, par exemple, ceci signifie que cinq expériences différentes ont été réalisées : la première pour mesurer le temps requis pour ajouter un nombre croissant d'actions d'un octet à une séquence, la seconde pour l'ajout d'actions de 500 octets à une autre séquence, et ainsi de suite. Les tailles d'actions prises en compte dans cette mesure vont de 1 octet à 100 Ko, ce qui est représentatif des cas d'utilisation des séquences de pair-actions que nous envisageons, en intégrant des petites actions comme les actions ι (introduites dans la Section 5.4) ou des actions plus importantes comprenant plusieurs lignes de texte.

La première observation qui peut être faite à partir de ces figures est que, à l'instar d'Artichoke-X et pour une taille d'action fixe, le temps pour ajouter des actions ou vérifier une séquence est linéaire par rapport au nombre d'actions. En l'occurrence, on constate qu'il faut en moyenne 0,12 ms pour ajouter une action d'un octet, là où il fallait 3,7 ms avec Artichoke-X : l'ajout d'action est donc environ 30 fois plus rapide avec Go-Artichoke. On peut observer cependant que cette performance est quasiment identique pour les séquences contenant des actions allant jusqu'à 1 Ko (les courbes correspondant à ces séquences sont presque parfaitement confondues). La séquence contenant les actions de 10 Ko présente un résultat légèrement plus lent avec une moyenne de 0,16 ms par ajout. Enfin, c'est avec la



(a) Ajout d'actions à la séquence.



(b) Vérification de la séquence.

FIGURE 6.14 : Temps d'exécution de Go-Artichoke avec une séquence de taille croissante selon plusieurs tailles d'action : (a) pour ajouter des actions dans la séquence et (b) pour vérifier la séquence (échelle logarithmique sur les ordonnées) © Quentin Betti.

séquence d'actions de 100 Ko que l'on peut noter une réelle différence, avec une moyenne de 0,56 ms par ajout. Ceci signifie que, même avec des actions 100 000 fois plus grandes, Go-Artichoke est tout de même près de 7 fois plus rapide qu'Artichoke-X.

La rapidité de l'ajout d'actions est à contraster avec le temps requis par la vérification de séquence. On constate en effet qu'il faut en moyenne 0,12 ms pour vérifier une pair-action contenant une action d'un octet, contre 0,1 ms pour Artichoke-X. La vérification est donc environ 1,2 fois plus rapide avec Artichoke-X. Cependant, on peut observer que, là aussi, les temps nécessaires pour vérifier des séquences de pair-actions contenant à l'origine des actions dont la taille varie d'un octet à 1 Ko sont quasiment identiques. Les séquences d'actions de 10 Ko présentent un temps de vérification moyen de 0,22 ms par pair-action, et celles de 100 Ko un temps moyen de 0,58 ms par pair-action. Ainsi, vérifier des séquences de pair-actions de 100 Ko avec Go-Artichoke prend seulement 6 fois plus de temps que la vérification de séquences contenant des actions d'un octet, donc 10^5 fois plus petites, avec Artichoke-X.

En outre, contrairement à Artichoke-X où le temps d'ajout était près de 38 fois plus important que le temps de vérification, Go-Artichoke présente des résultats similaires pour les deux opérations, avec un ajout 30 fois plus rapide et une vérification *très* légèrement plus lente qu'Artichoke-X. En réalité, on pouvait s'attendre à de tels résultats, et ce pour plusieurs raisons. Premièrement, selon plusieurs tests (Radadiya, 2019; Benchmarks Game, 2020; Cook, 2015), un programme Go est globalement un peu plus rapide que son équivalent en Java. Ceci est probablement dû au fait que les programmes Java sont exécutés par l'intermédiaire d'une machine virtuelle, ce qui n'est pas le cas de Go. De plus, AES et ECDSA sont plus rapides que RSA pour, respectivement, le chiffrement et la génération de signatures. Il était donc probable que ce gain de vitesse surpasse le « ralentissement » induit par le hachage avec Keccak-256 au lieu de MD5. À contrario, ECDSA est plus lent que RSA pour la vérification de signature,

ce qui explique la légère augmentation de temps requise par la vérification des séquences de pair-actions.

La Figure 6.15, quant à elle, montre l'évolution de l'espace nécessaire pour stocker une séquence de pair-actions selon, là aussi, un nombre d'actions croissants et des tailles d'action variées. Sans surprise, à l'instar d'Artichoke-X et pour une taille d'action fixe, la mémoire requise pour stocker la séquence de pair-actions est linéaire par rapport au nombre d'actions. Pour des actions d'un octet, on observe que 267 octets par pair-action sont nécessaires, contre 930 octets pour Artichoke-X : Go-Artichoke est donc 3,5 fois plus économe en stockage pour des actions d'un octet. La courbe concernant les actions de 256 octets (la taille maximale des actions dans Artichoke-X) n'est pas représentée ici, mais les résultats indiquent 720 octets par pair-action, ce qui reste 1,3 fois plus économe. Là aussi, nous pouvions nous attendre à de tels résultats pour deux raisons en particulier. Premièrement, les signatures ECDSA sont généralement plus petites que celles produites par RSA (à niveau de sécurité égale) : nous générons donc des signatures de 65 octets avec Go-Artichoke, au lieu de 256 octets pour Artichoke-X. Deuxièmement, nous avons mentionné dans la Section 6.2.3 que pour des tailles d'actions variant entre 1 et 256 octets, le chiffrement par RSA avec des clés de 2048 bits produisait toujours un cryptogramme de 256 octets. À contrario, avec AES la taille du cryptogramme est la même que celle de la donnée [55], expliquant ce gain important d'espace pour des actions d'un octet.

Sur cette figure, on peut cependant constater que la mémoire occupée par une séquence de pair-actions n'est pas linéaire par rapport à la taille des actions. En effet, pour un nombre

55. En pratique, il faut ajouter à la taille du cryptogramme celles des IV et AT correspondants au chiffrement par AES-GCM. Or, comme nous l'avons dit précédemment, nous utilisons des IV et AT de respectivement 96 et 128 bits. Le résultat total du chiffrement par AES-GCM est donc plus grand de 224 bits par rapport à la donnée originelle.

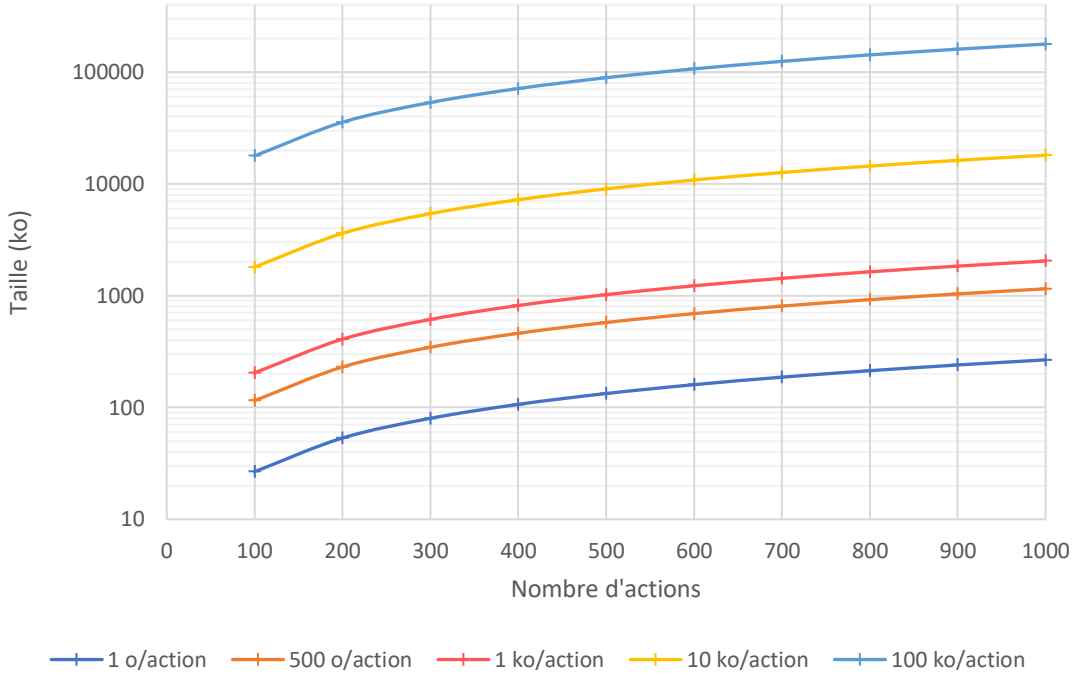


FIGURE 6.15 : Mémoire requise par Go-Artichoke pour une séquence de pair-actions de taille croissante selon plusieurs tailles d'action (échelle logarithmique en ordonnées) © Quentin Betti.

d'actions fixe, on pourrait s'attendre à ce qu'une séquence de pair-actions de 1 ko soit 1 000 fois plus grande qu'une séquence de pair-actions de 1 octet, mais en réalité elle n'est que 7,7 fois plus volumineuse. Cette différence s'explique par le fait que la taille d'une pair-action n'est pas strictement proportionnelle à la taille de son action originelle. À la taille du cryptogramme AES de l'action, il faut en effet ajouter celles des paramètres du mode GCM (c.-à-d. l'IV et l'AT correspondant) et du condensat, ainsi que celles des identifiants du pair et du groupe. À l'exception du cryptogramme, tous ces éléments ont une taille fixe, que l'action soit d'1 octet ou de plusieurs kilooctets. Ainsi, la surcharge relative de mémoire requise dépend de la taille des actions.

Définition 11 (Surcharge relative)**.** *Soient $l_{\overline{s}^*}$ et $l_{actions}$ les tailles en mémoire respectivement occupées par une séquence de pair-actions $\overline{s}^*$ et l'ensemble cumulé des actions à insérer dans $\overline{s}^*$, la surcharge en mémoire relative $\eta_{\overline{s}^*}$ provoquée par $\overline{s}^*$ est définie de la façon suivante :*

$$\eta_{\bar{s}^*} = \frac{l_{\bar{s}^*} - l_{actions}}{l_{actions}} = \frac{l_{\bar{s}^*}}{l_{actions}} - 1$$

La figure Figure 6.16 montre l'évolution de la surcharge relative selon la taille des actions. Comme nous l'avons expliqué, on peut constater qu'une séquence de pair-actions constituée d'actions d'un seul octet induit une surcharge relative très importante : la séquence de pair-actions occupe environ 267 fois plus de mémoire que la totalité des actions originelles. Cependant, cette surcharge diminue lorsque la taille des actions augmente, jusqu'à atteindre un palier. Ainsi, les séquences de pair-actions de 10 Ko et plus provoquent une surcharge d'environ 78%. Cette diminution s'explique par le fait que les éléments de taille fixe des pair-actions (c.-à-d. l'IV, l'AT, le condensat et les identifiants) deviennent négligeables par rapport aux actions chiffrées. Pour des tailles d'actions très grandes (à partir de 10 Ko d'après la figure), la surcharge relative peut donc être réduite au surplus provoqué par le double encodage en base 64 des données. Un tel encodage résulte en une augmentation de la taille des données d'environ $\frac{4}{3}$. Puisque nous avons deux encodages successifs, ce ratio est donc de $\frac{16}{9} \approx 1,78$ ce qui, relativement à la taille initiale des actions, donne bien une surcharge relative d'environ 0,78.

De manière générale, les résultats de cette figure confirment que, du point de vue de la mémoire utilisée, il vaut mieux stocker de grandes actions plutôt que des petites actions. Par exemple, dans un scénario médical, si l'infirmière se rend compte qu'un traitement provoque plusieurs effets indésirables à un patient, il est préférable qu'elle ajoute tous ces effets dans une même action plutôt que de le faire en plusieurs fois. Cet argument est également valable en termes de temps de traitement puisque, comme nous pouvons le voir dans la Figure 6.14, le temps nécessaire pour ajouter ou vérifier 1 000 actions d'un octet est bien plus important que pour une seule action d'un kilooctet, alors que la quantité d'informations reste la même.

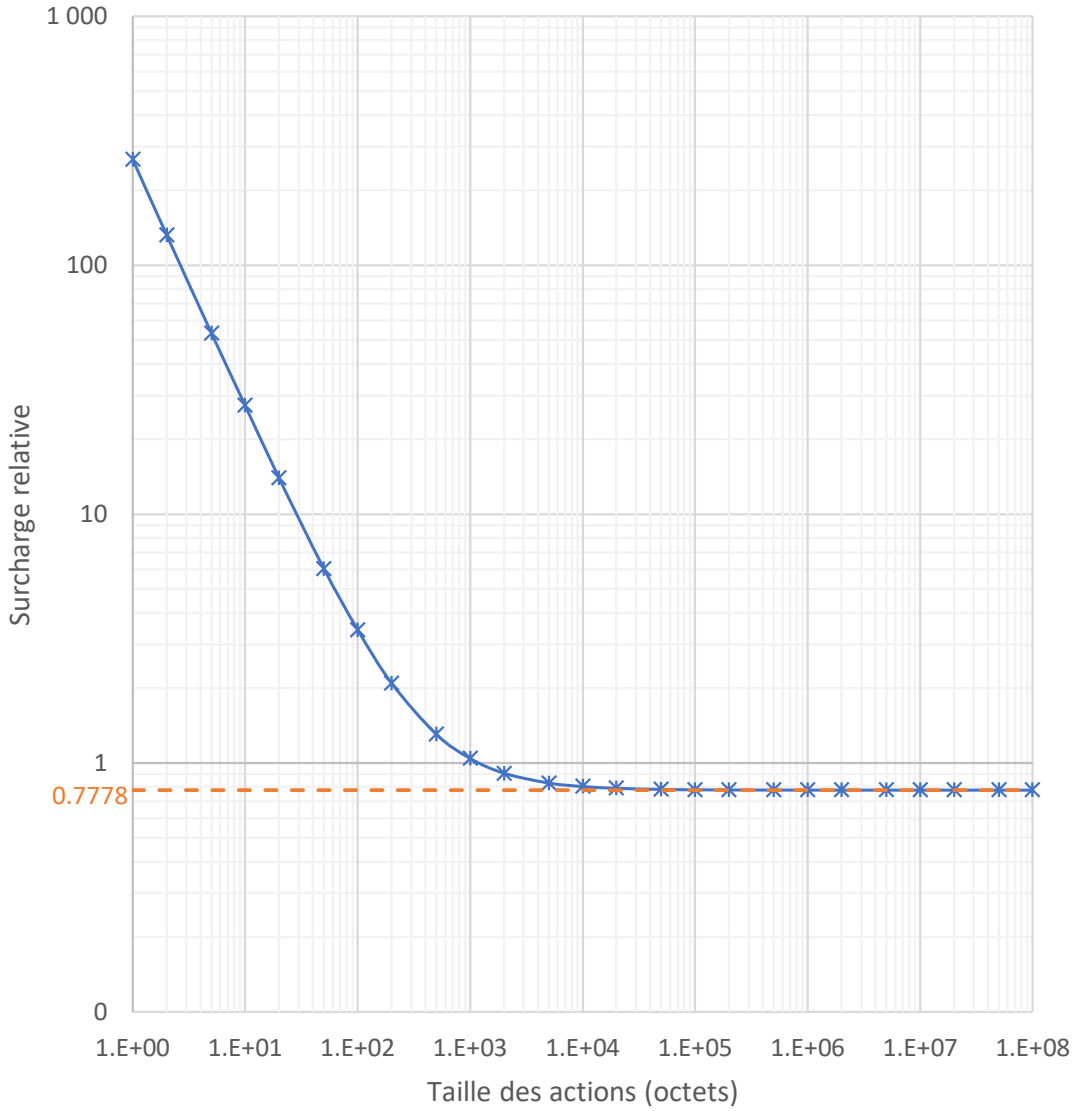


FIGURE 6.16 : Surcharge relative de la mémoire requise pour des séquences de pair-actions de taille d'actions croissante (échelle logarithmique en abscisses et ordonnées) © Quentin Betti.

Ainsi, lorsqu'un pair réalise plusieurs actions successives sur le document au nom d'un même groupe, il pourrait être envisagé de les rassembler en une seule et même pair-action.

6.3.3 STOCKAGE DES CLÉS AVEC ARTICHOKE-KEYRING

En s'appuyant sur la définition et l'implémentation du portefeuille Ethereum (Nakov, 2018), nous avons conçu et développé une bibliothèque qui permet à un pair de sauvegarder de manière sécurisée sa paire de clés ECDSA publique/privée et les clés AES des différents groupes auxquels il appartient. Cette bibliothèque, Artichoke-Keyring, est écrite en Java et disponible sous licence open source en téléchargement [56].

À l'origine, un portefeuille Ethereum contient seulement l'adresse et la clé privée chiffrée d'un compte Ethereum (on connait également ce type de fichier sous le nom de *porte-clés*). Un exemple de ce type de fichier JSON est donné en Figure 6.17. Parmi les champs qui le composent, on peut noter le champ `address` qui, comme son nom l'indique, contient l'adresse du compte Ethereum concerné, et le champ/objet `Crypto` qui va contenir toutes les informations nécessaires au déchiffrement de la clé privée.

Pour expliquer son fonctionnement, il est important de préciser que le portefeuille Ethereum est protégé par mot de passe. La première étape que l'utilisateur doit effectuer pour récupérer sa clé privée est donc de renseigner son mot de passe. À partir de ce dernier et d'un sel cryptographique, le client Ethereum va utiliser une *fonction de dérivation de clé* (abrégée KDF, pour *Key Derivation Function*) pour générer une clé de 256 bits. La KDF utilisée ici est `scrypt` (Josefsson & Percival, 2016), dont les différents paramètres sont spécifiés dans le champ `kdfparams`. Parmi ceux-ci, on note les paramètres `dklen` et `salt` qui indiquent,

56. https://github.com/qbetti/artichoke-keyring

```
{
  "version": 3,
  "id": "07a9f767-93c5-4842-9afd-b3b083659f04",
  "address": "aef8cad64d29fcc4ed07629b9e896ebc3160a8d0",
  "Crypto": {
    "ciphertext": "99d0e66c67941a08690e48222a58843ef2481e110969...",
    "cipherparams": { "iv": "7d7fabf8dee2e77f0d7e3ff3b965fc23" },
    "cipher": "aes-128-ctr",
    "kdf": "scrypt",
    "kdfparams": {
      "dklen": 32,
      "salt": "85ad073989d461c72358ccaea3551f7ecb8e672503cb...",
      "n": 8192,
      "r": 8,
      "p": 1
      },
    "mac": "06dcf1cc4bffe1616fafe94a2a7087fd79df444756bb..."
  }
}
```


FIGURE 6.17 : Exemple de fichier portefeuille Ethereum (certaines valeurs sont tronquées), tiré de Nakov (2018). Image sous licence MIT.

```
{
  "salt": "97379c876727b623f3cc871a178ba0de68a0ba66f890...",
  "publicKey": "02abf5269de09a327c7f1bd1089ef5cc2a78365f04c4...",
  "privateKey": {
    "cipherText": "c28aa297aa3b1360560363eadeeb7d6bbb7a8af8adf4...",
    "iv":"db2d93abf11ae6931cea7f0fb8e73e15"
  },
  "groups": [{
      "id": "G1",
      "cipherText":"387bb587daaae89999b940d16d62533ad9780684c85f...",
      "iv":"b3991341c0e35b0c140d82d38de0f205"
    }, {
      "id": "G2",
      "cipherText": "d45f9a7ac5c08d5961ad2e523caa2886fa06d0a635f...",
      "iv": "aaaaac00dc8d0fcb2d5205830df8cbfa"
  }]
}
```


FIGURE 6.18 : Exemple de fichier porte-clés généré par Artichoke-Keyring (certaines valeurs sont tronquées) © Quentin Betti.

respectivement, la taille de la clé à générer (en octets) et le sel cryptographique utilisé pour la génération. Une fois la clé générée, celle-ci est utilisée dans l'algorithme de chiffrement identifié dans le champ `cipher` qui, dans notre exemple, est AES en mode *Counter* (CTR) (Dworkin, 2001) ; on peut alors déchiffrer le cryptogramme de la clé privée présent dans `ciphertext` à l'aide de l'IV défini dans `cipherparams`.

Pour résumer, le mot de passe est utilisé pour générer, via `scrypt`, une clé AES qui permet de déchiffrer le cryptogramme de la clé privée. Une erreur dans le mot de passe créerait une clé AES complètement différente et donc ne permettrait pas de récupérer la clé privée. De plus, il est important de noter que `scrypt` est une fonction volontairement *lente*, afin d'éviter les attaques par force brute. Enfin, il est nécessaire que le mot de passe soit fort et gardé secret à tout prix puisque toute la sécurité du portefeuille repose sur la connaissance du mot de passe.

Dans Artichoke-Keyring, nous modifions légèrement cette structure pour pouvoir stocker à la fois la clé privée et les clés des groupes d'un pair. Un exemple d'un porte-clés généré avec Artichoke-Keyring est donné en Figure 6.18 pour en illustrer la structure. Une des différences les plus visibles est que tous les paramètres spécifiques aux différents algorithmes utilisés (*p. ex.*, tailles des clés, nom des algorithmes, modes) ne sont pas enregistrés dans le fichier mais sont plutôt directement renseignés dans le code de la bibliothèque ; ceci a pour but de simplifier la structure des fichiers porte-clés. En l'occurrence, nous utilisons les mêmes mécanismes que les portefeuilles Ethereum, à savoir `scrypt` (avec les mêmes paramètres) pour la génération de clé et AES-CTR pour le chiffrement. Les champs restants sont alors les suivants :

— `publicKey` : la clé publique du pair, qui sert d'identifiant dans les séquences de pair-actions générées avec Go-Artichoke.

— `salt` : le sel cryptographique utilisé pour la génération de la clé via `scrypt`.

— `privateKey` : objet contenant les données utiles au déchiffrement de la clé privée, c.-à-d. son cryptogramme (`ciphertext`) et l'IV associé (`iv`).

— `groups` : tableau contenant les informations concernant les clés des groupes. Chaque élément est composé de l'identifiant du groupe (`id`), le cryptogramme de la clé symétrique du groupe (`cipherText`) et l'IV associé (`iv`).

À l'instar des portefeuilles Ethereum, le fichier est protégé par mot de passe. Lorsque le pair le renseigne, le mot de passe est utilisé avec le sel cryptographique par la fonction `scrypt` qui génère une clé AES de 256 bits. Cette clé permet ensuite de déchiffrer la clé privée ainsi que les clés des groupes, en combinaison avec les IV correspondants.

Ainsi, la bibliothèque Artichoke-Keyring permet à un pair de créer un nouveau porte-clés en générant une nouvelle paire de clés ECDSA et en choisissant un mot de passe pour chiffrer sa clé privée. Il est ensuite possible d'ajouter des groupes en renseignant leur identifiant et la

clé symétrique associée ; la bibliothèque s'occupe de chiffrer les clés de groupe avec la clé générée via `scrypt`. Elle permet également de mettre à jour les informations des groupes en ajoutant des clés, ou en les modifiant en cas de changement. Artichoke-Keyring fournit donc une interface facile d'utilisation pour stocker les clés des pairs de façon sécurisée. On peut noter que ce type de fichier peut être conservé localement mais qu'il pourrait tout à fait être stocké sur un serveur distant sans que sa sécurité en soit diminuée, étant donné que seul le bon mot de passe peut débloquer le porte-clés.

6.4 COMPARAISON ENTRE LES BLOCKCHAINS ET LES SÉQUENCES DE PAIR-ACTIONS

Maintenant que nous avons détaillé les résultats expérimentaux, il est temps de conclure sur une comparaison des deux approches. Celle-ci se base sur plusieurs critères qui nous semblent pertinents vis-à-vis de la construction d'historiques sécurisés d'artefacts et de la vérification de propriétés de cycles de vie. Ces critères couvrent des domaines variés (*p. ex.*, sécurité, fonctionnalités, performances) et sont résumés dans le Tableau 6.2 ; chacun d'entre eux est détaillé dans une sous-section associée, précisée dans la dernière colonne du tableau. À ces critères de comparaison succède une discussion plus générale sur l'applicabilité des deux solutions, permettant de conclure sur l'utilisation générale de ces deux approches.

6.4.1 GESTION DES PAIRS, CONFIDENTIALITÉ ET CONTRÔLE D'ACCÈS

Dans un réseau Ethereum, toute personne ayant créé une adresse Ethereum et la paire de clés correspondante peut participer en envoyant des transactions au réseau. On peut restreindre cet accès en déployant un réseau *privé*, séparé des réseaux publics traditionnels d'Ethereum. La connexion à ce réseau pourrait alors s'effectuer via un proxy qui serait chargé d'appliquer les différentes politiques d'accès au réseau. Cependant, une fois connecté à celui-ci, un

Critère	Blockchain Ethereum	Séquence de pair-actions	Section
Gestion des pairs, confidentialité et contrôle d'accès	Tout utilisateur peut créer un compte, interagir avec la blockchain et en extraire les transactions.	Le pair doit être associé à au moins un groupe pour écrire une action, et les actions sont chiffrées.	6.4.1
Gestion des clés et identités	Intégrée seulement pour les clés publiques. Identité et échange de clés possibles via *smart contracts*.	Via un annuaire externe. Échange de clé possible via les séquences.	6.4.2
Spécification et application de cycle de vie	Ici, externes à la blockchain. Pourraient être directement intégrées dans des *smart contracts*.	Externes, mais pourraient être directement intégrées sous forme d'actions dans la séquence.	6.4.3
Performances	$\approx$ 15 TPS actuellement pour *l'ensemble du réseau*.	Ajout et vérification : $\approx$ 4 200 actions par seconde pour *un seul ordinateur*.	6.4.4
Surcharge en mémoire relative (100 octets/action)	$\eta_{ETH} = 15,5$	$\eta_{\bar{s}^*} = 3,4$	6.4.5
Isolation des artefacts	Non	Oui	6.4.6
Actions concurrentes	Oui	Non	6.4.7
Sécurité de l'historique	Nécessite un consensus et que tous les clients gardent une copie locale.	Basée uniquement sur les signatures et les fonctions de hachages. Peut entraîner des problèmes de duplication/troncature des historiques.	6.4.8

TABLEAU 6.2 : Tableau comparatif entre les approches par blockchain et par séquence de pair-actions © Quentin Betti.

pair possédant un client Ethereum peut avoir accès à toutes les transactions de la blockchain, qu'elles le concernent ou non. Ce principe est fondamental dans la philosophie de la blockchain, puisqu'il procure une transparence complète à tous les utilisateurs. Ceci n'est sans doute pas souhaitable dans certains secteurs tels que le domaine médical (*p. ex.*, les informations sur la santé du patient ne devrait être révélées qu'au personnel compétent) et la logistique (*p. ex.*, un transporteur acheminant un colis entre deux *hubs* ne devrait pas forcément avoir accès au contenu du colis en question et à l'identité de son émetteur ou destinataire). Si l'on souhaite qu'une information contenue dans la blockchain soit réservée à un groupe spécifique d'utilisateurs, il est donc nécessaire de chiffrer cette donnée avant qu'elle soit incluse dans une transaction.

À cet égard, les séquences de pair-actions offrent une alternative intéressante. Pour pouvoir participer à l'élaboration de séquence de pair-actions d'un artefact, un pair doit nécessairement posséder sa propre paire de clés publique/privée et appartenir à au moins un groupe. Les actions qu'il entreprendra seront chiffrées avec les clés de groupe correspondantes, instaurant ainsi un contrôle d'accès natif. En effet, accéder à une action nécessite d'appartenir au groupe au nom duquel celle-ci a été effectuée et, en outre, il est seulement possible d'ajouter les actions autorisées pour ce groupe [57]. Ainsi, si un pair participant à la manipulation de l'artefact souhaitait interagir avec des actions qui ne concernent pas les groupes dont il est membre, ses capacités d'accès en écriture/lecture seraient, comme pour un individu extérieur au processus, complètement nulles. Dans ce contexte, on peut dire que les séquences de pair-actions offrent une protection contre des accès illicites à la fois externes et internes au processus.

57. Il serait en soit possible pour le pair d'ajouter tout de même la pair-action à la séquence, mais les autres pairs seraient alors capables de constater l'ajout d'une action non-autorisée pour ce groupe et d'identifier le responsable de cette tentative.

Il est d'ailleurs important de noter que, même si nos formalisations et implémentations des séquences de pair-actions suggèrent de chiffrer totalement les actions, il est possible de *moduler* leur confidentialité. Dans certains contextes, il pourrait être en effet préférable de ne chiffrer qu'une partie de l'action. Par exemple, si l'action est définie comme un triplet $\langle \bar{\sigma}, t, v \rangle$, conformément à la formalisation de la Section 5.1.2, il pourrait être décidé de ne chiffrer que la valeur v de l'action, et de laisser le document concerné $\bar{\sigma}$ et le type de l'action t en clair. Ce faisant, les pairs extérieurs au groupe concerné pourrait voir qu'une action d'un certain type a été effectuée sur un document précis, sans connaître les détails de l'opération, dans lesquels résident probablement les informations sensibles. Le chiffrement des pair-actions est ainsi adaptable aux différents cas d'utilisation.

6.4.2 GESTION DES CLÉS ET IDENTITÉS

Les deux approches font intervenir des clés cryptographiques. Ethereum et Go-Artichoke utilisent des paires de clés publiques/privées ECDSA, en plus de clés symétriques AES pour les groupes des séquences de pair-actions. Cependant, nous avons vu dans la section précédente qu'un chiffrement externe était nécessaire si l'on souhaitait conserver la confidentialité des données stockées dans Ethereum. À fonctionnalités équivalentes, on peut donc dire que les deux solutions nécessitent de gérer les mêmes types de clés.

Dans Ethereum, il n'existe pas de registre où toutes les adresses et clés publiques de compte seraient répertoriées. Plutôt, on peut récupérer une adresse directement depuis une transaction de la blockchain ; dans la plupart des cas, cependant, l'utilisateur communique son adresse par lui-même aux autres utilisateurs via, *p. ex.*, un site internet personnel, un code QR généré par un terminal de paiement (BitPay, 2020) ou une application spécifique (Coinbase Commerce, 2020; CoinGate, 2020). Dans le cadre de processus opérationnels, on pourrait imaginer qu'une entreprise mette en place un annuaire identifiant les clés et adresses

des employés. Ceci peut cependant être directement intégré dans la blockchain, à travers un *smart contract* qui conserverait une liste de paires nom-adresse et qui offrirait une interface permettant de mettre à jour cette liste, d'identifier le possesseur d'une adresse et inversement.

De façon similaire aux transactions blockchains et aux adresses Ethereum, chaque pair-action d'une séquence construite par Go-Artichoke identifie le responsable de l'action par sa clé publique. Cette structure pourrait être modifiée pour que cet identifiant fasse un lien plus évident avec l'identité réelle de la personne, en concaténant par exemple ses nom et prénom à la clé publique. Cependant, dans un tel système, il serait aisé d'apposer le nom d'un autre utilisateur à côté de sa propre clé publique : un pair se reposant uniquement sur le nom renseigné pour identifier l'utilisateur serait alors trompé. Un système externe d'annuaire (qu'il soit en ligne ou local) est donc nécessaire pour garder l'identité des pairs et leur clé publique à jour.

Concernant les clés privées, nous avons montré dans la Section 6.3.3 comment des systèmes de portefeuilles et de porte-clés étaient respectivement implémentés dans les clients Ethereum et la bibliothèque Artichoke-Keyring. De façon générale, les clés privées sont conservées dans un fichier protégé par mot de passe ; ce fichier peut être lui-même stocké en ligne ou localement. Cependant, Artichoke-Keyring permet également de stocker les clés secrètes des groupes d'un pair. Ces clés doivent lui être communiquées au préalable par les membres des groupes concernés ou par un « administrateur » (humain ou logiciel) détenant l'ensemble des clés. Cette dernière solution est en réalité peu recommandable étant donné qu'une compromission de cet administrateur entrainerait le vol de toutes les clés de groupes, et donc la compromission totale des historiques d'artefacts impliqués. Néanmoins, nous avons mentionné dans la Section 5.3 que la séquence de pair-actions pouvait éventuellement servir d'intermédiaire pour s'échanger de nouvelles clés. De la même façon, avec Ethereum les clés symétriques nécessaires au chiffrement des actions pourraient être partagées directement via

des *smart contracts* en utilisant, par exemple, des protocoles d'échange de clés tels que Diffie-Hellman (Diffie & Hellman, 1976) et son application aux courbes elliptiques *Elliptic-curve Diffie-Hellman* (Barker *et al.*, 2006), abrégée ECDH.

6.4.3 SPÉCIFICATION ET APPLICATION DE CYCLE DE VIE

Dans le Chapitre 4, nous avons concrètement spécifié et implémenté des propriétés de cycle de vie à l'aide de BeepBeep et de la palette permettant l'interaction avec des blockchains Ethereum. Si les événements alimentant les moniteurs de ces propriétés sont directement issus de la blockchain, la vérification reste *externe* : elle n'est pas directement réalisée par le réseau Ethereum, mais plutôt par des moniteurs qui observent celle-ci. De même pour les séquences de pair-actions, pour lesquelles nous avons pu définir et formaliser des cycles de vie dans le Chapitre 5 : la vérification n'est pas réalisée par la séquence elle-même, ce sont les pairs qui, lorsqu'ils reçoivent l'artefact, sont responsables de vérifier le bon respect du cycle de vie.

Ceci a divers avantages. Tout d'abord, l'externalisation du procédé permet l'indépendance de l'historique vis-à-vis de la spécification des propriétés et de leur vérification. Ce faisant, un changement dans cette spécification ou dans la façon de la vérifier n'impacte ni la construction ni la circulation de l'historique. Par exemple, si une erreur s'est glissée dans la mise en place du cycle de vie à respecter, celui-ci peut être changé tout en conservant l'historique intact. De même, pour des raisons technologiques, chacun est libre de vérifier le cycle de vie avec l'outil qu'il souhaite ; qu'il s'agisse d'un programme différent ou qu'il soit exécuté sur un autre système d'exploitation, l'historique est utilisable par tous, à partir du moment où le programme choisi est capable de *lire* ce dernier. Il s'agit d'une philosophie semblable à celle, par exemple, des API REST (Fielding & Taylor, 2002) fournies par certains serveurs web et qui n'imposent *a priori* pas ou peu de contraintes sur la technologie utilisée par le client. Dans ce contexte, l'historique est comparable aux données envoyées sous un

certain format par ces API, tandis que la vérification se rapproche du traitement de ces données par les programmes clients.

Le second argument concerne les performances de traitement. Qu'il s'agisse du nombre de TPS supporté par le réseau Ethereum, ou le nombre d'actions par seconde qu'il est possible d'ajouter à une séquence de pair-actions, si la vérification du cycle de vie avait lieu directement en interne ceci aurait probablement un impact négatif notable sur leur capacité de traitement. Or, comme nous l'avons vu dans l'exemple consacré à la logistique, la capacité du système en termes de fréquence d'ajout de nouveaux événements peut représenter un critère crucial selon les cas d'utilisation. Apporter une charge supplémentaire au traitement est donc peu recommandable, d'où l'intérêt de reporter cette charge sur un système externe, exécuté en parallèle, qui permettrait tout de même d'effectuer la vérification en temps réel si ceci est nécessaire.

Il est néanmoins possible d'implémenter cette vérification en interne de la blockchain Ethereum assez facilement. Il suffit en effet de déployer des *smart contracts* qui agiraient tels des moniteurs et par lesquels toute action devrait transiter. Ceci implique évidemment que le cycle de vie soit spécifiable, et éventuellement modifiable, directement par les contrats. Ainsi, la spécification et la vérification du cycle de vie aurait lieu exactement en même temps que l'ajout des actions, ce qui offre la possibilité de produire des moniteurs ultra-réactifs, capables de déclencher des alertes au moment exact de l'occurrence d'une action suspecte [58]. Les séquences de pair-actions, dans leur implémentation actuelle, ne disposent pas de cette fonctionnalité puisque ceci nécessite qu'il y ait une certaine forme de *synchronisation* entre les pairs, ce qui n'est présentement pas le cas (ce point est détaillé dans la Section 6.4.7). On peut cependant noter que le cycle de vie pourrait être intégré directement dans l'artefact, *p. ex.*

58. Bien entendu, nous mettons ici de côté le temps de latence qui serait lié aux performances maximales des réseaux Ethereum en termes de TPS et à la synchronisation des clients en général.

sous la forme de méta-données, et qu'une pair-action contenant sa spécification sérialisée soit ajoutée à l'historique lors de la création du document. Ainsi, tous les pairs peuvent avoir connaissance du cycle de vie à travers l'historique lui-même, et les modifications potentielles du cycle pourraient être directement réalisées via des pair-actions effectuées ultérieurement par des pairs autorisés.

6.4.4 PERFORMANCES

La capacité du système à traiter un certain nombre d'actions en un temps donné est un des critères de comparaison les plus importants. En effet, certains cas d'application (tels que celui décrit plus tôt dans le cadre de la logistique hyperconnectée) nécessitent que le réseau soit en mesure de traiter plusieurs centaines de milliers d'événements par seconde. Cet ordre de grandeur est pour l'instant inaccessible pour les implémentations de réseaux de blockchains actuelles, où des milliers d'ordinateurs (qui sont parfois très haut de gamme, notamment pour les nœuds mineurs) peinent à dépasser les dizaines de TPS. Ethereum par exemple, permet actuellement de traiter seulement une quinzaine de transactions par seconde. Nous avons vu que certains réseaux promettaient des taux de TPS bien plus élevés, mais ces derniers restent théoriques (*p. ex.*, Futurepia et EOS, avec leurs promesses respectives de 300 000 et un million de TPS).

Une autre évolution intéressante est celle d'Ethereum même, avec l'arrivée imminente d'Ethereum 2.0 (Ethereum Foundation, 2020), le début de la transition étant prévue pour 2021. Cette amélioration est basée sur deux changements majeurs. Premièrement, Ethereum 2.0 utilisera un consensus de type Proof-of-Stake au lieu du Proof-of-Work actuel, ce qui permettra d'améliorer grandement la vitesse et la consommation énergétique du réseau. Deuxièmement, cette nouvelle version introduit la notion de *fragments* (*shards* en anglais). Actuellement, chaque nœud du réseau contient l'entièreté de l'état global de la blockchain et traite toutes les

transactions ; Ethereum 2.0 se démarque de son prédécesseur en partitionnant la blockchain en plusieurs fragments, où des groupes de nœuds sont aléatoirement assignés au stockage de certains comptes et à la vérification des transactions qui concernent uniquement ces derniers. D'après Vitalik Buterin, le créateur d'Ethereum, ces modifications permettraient d'envisager les centaines de milliers de TPS, voire potentiellement quelques millions. Là encore, cependant, il s'agit d'estimations théoriques.

Les séquences de pair-actions, quant à elles, permettent à un ordinateur portable *modeste* de traiter (ajout + vérification) près de 4 200 actions par seconde. Outre le fait que cette performance est déjà bien supérieure à celle de l'implémentation actuelle du réseau entier d'Ethereum, il est important de noter que les séquences de pair-actions traitent chaque artefact indépendamment, là où les blockchains réunissent toutes les transactions du réseau dans un même registre. Ainsi, il est tout à fait possible d'assigner le traitement d'artefacts distincts à plusieurs ordinateurs. Sommairement, si des artefacts occupent déjà complètement la capacité de traitement d'un ordinateur, il suffit d'attribuer les autres artefacts à un second ordinateur, et donc potentiellement doubler le nombre d'actions qu'il est possible de traiter par seconde. Si on répartit parfaitement le traitement des artefacts à travers 8 000 ordinateurs (le nombre moyen de clients Ethereum), on peut donc potentiellement atteindre $6\,000 \times 4\,200 = 33,6$ millions d'événements par seconde, voire plus avec des ordinateurs haut de gamme. Le taux d'actions par seconde pourrait donc théoriquement évoluer linéairement avec le nombre d'ordinateurs alloués au traitement, ce qui n'est absolument pas le cas d'Ethereum actuellement.

6.4.5 SURCHARGE EN MÉMOIRE RELATIVE

Un autre critère de comparaison essentiel est la surcharge en mémoire relative, c.-à-d. la mémoire totale occupée par une transaction (dans le cas des blockchains) ou d'une pair-action (dans le cas des séquences de pair-actions) par rapport à la taille de la donnée initiale à stocker.

À titre d'exemple, considérons les transactions stockées dans la blockchain Ethereum de notre simulation de logistique hyperconnectée. Les actions de base envoyées par les agents AnyLogic aux nœuds Ethereum (détaillées dans la Section 4.2.2), pèsent en moyenne 97 octets par action. Or, comme nous l'avons vu en Section 6.1.2, la taille finale d'une transaction (contenant une seule action) est d'environ 1,6 Ko. Ceci signifie que la surcharge relative en mémoire de la blockchain Ethereum est donc :

$$\eta_{ETH} = \frac{1\,600}{97} - 1 = 15,5$$

Du côté des séquences de pair-actions, si nous nous référons à l'abaque en Figure 6.16 montrant l'évolution de la surcharge relative provoquée par Go-Artichoke en fonction de la taille des actions, on peut constater que pour des actions d'environ 100 octets, la surcharge relative d'une séquence de pair-actions $\overline{s}^*$ est $\eta_{\overline{s}^*} = 3,4$. Dans cet exemple, on a donc une surcharge relative près de 5 fois moins importante avec les séquences de pair-actions par rapport à la blockchain Ethereum, ce qui est loin d'être négligeable, particulièrement si l'on traite des volumes de données très importants ou si les historiques sont stockés dans des systèmes limités en capacité de stockage, tels que des appareils IoT.

6.4.6 ISOLATION DES ARTEFACTS

Ce qui différencie principalement les blockchains et les séquences de pair-actions réside dans la philosophie de gestion des historiques et des artefacts. Dans le cas de la blockchain, toutes les transactions sans exception sont traitées et stockées par chaque nœud du réseau. Ces transactions, potentiellement, n'ont aucun rapport entre elles, et certaines n'intéressent que quelques pairs, mais elles seront tout de même conservées par l'ensemble du réseau dans la même blockchain. Dans un premier temps, cela entraîne une duplication massive des données,

dont le seul intérêt est d'assurer la sécurité de l'historique. Deuxièmement, si un ensemble de transactions est jugé obsolète et inutile à ce jour, il est impossible de le supprimer, puisque les condensats des blocs seraient alors incorrects, remettant en cause un des principes de sécurité fondamentaux de la blockchain. De même pour des blocs entiers, qui ne peuvent être supprimés sans mettre en péril le principe de chaînage de la blockchain. Une information ne peut donc pas être effacée, et la blockchain est vouée à croître indéfiniment [59].

Les séquences de pair-actions résolvent ce problème en isolant complètement les artefacts les uns des autres. Un séquence est propre à un unique artefact, et ne dépend donc *a priori* d'aucun autre artefact ou séquence. Si des actions ont des effets sur plusieurs artefacts, alors il serait peut-être préférable de les stocker dans une séquence commune à ceux-ci. Les intérêts de cette solution sont multiples, notamment en termes de stockage et de performances du système. Premièrement, vis-à-vis des problèmes évoqués dans le paragraphe précédent, l'isolation des artefacts permet de supprimer complètement l'historique de l'un d'entre eux sans que cela affecte les autres. De plus, un pair n'a pas à récupérer et vérifier les séquences d'artefacts qui ne le concernent pas : il lui suffit de choisir ou de recevoir celles qui l'intéressent et ainsi il ne s'encombre pas avec des données inutiles. L'isolation des artefacts permet également l'indépendance de leur traitement, ce qui a un effet bénéfique sur les performances générales. En effet, comme nous l'avons vu en Section 6.4.4, un ordinateur peut être alloué au traitement d'un ensemble prédéfini d'artefacts et de leur séquence de pair-actions, lui évitant de dépenser des ressources calculatoires pour des artefacts qui ne le concernent pas [60].

---

59. C'est d'ailleurs une des raisons pour lesquelles il peut être si long de mettre en place un nouveau nœud Ethereum à l'heure actuelle. En effet, il est nécessaire de télécharger près de 160 Go de transactions (Blockchair, 2020), et chacune d'entre elles est téléchargée, vérifiée puis traitée individuellement par le nœud pour mettre à jour la copie locale de l'état de la blockchain. Le processus peut donc prendre en réalité plusieurs jours avec une connexion et un ordinateur moyens.

60. Cet aspect des séquences de pair-actions peut faire penser au mécanisme de fragmentation d'Ethereum 2.0, vu en Section 6.4.4, poussé à l'extrême. Néanmoins, pour des raisons de sécurité, dans Ethereum 2.0 les nœuds ne peuvent pas choisir les transactions qu'ils traitent et conservent : elles leur sont attribuées aléatoirement.

6.4.7 ACTIONS CONCURRENTES

Cette isolation a cependant un défaut non négligeable : dans leur version actuelle, les séquences de pair-actions ne permettent pas d'effectuer des actions concurrentes, c.-à-d. des actions réalisées par des pairs différents au même moment sur un artefact identique. En effet, la séquence de pair-actions attend par définition des actions séquentielles, puisque le condensat d'une pair-action est calculé via celui de la pair-action précédente. Si deux actions ont lieu en même temps, elles ont alors toutes les deux la même pair-action parente, et on ne peut choisir laquelle insérer en premier sans que la deuxième ait un condensat erroné. Dans la blockchain, ceci n'est pas un problème grâce aux consensus (qui sont d'ailleurs les principales raisons des faibles taux de TPS des réseaux courants). La séquence de pair-actions actuelle n'établit pas de consensus pour la simple raison que, pour l'instant, notre raisonnement repose uniquement avec des pairs « hors-ligne », qui réalisent les actions sur leur copie locale de l'artefact et la transmettent accompagnée de la séquence correspondante aux pairs suivants. Nous considérons en effet uniquement des cycles de vie où aucun *conflit* d'insertion d'actions ne devrait avoir lieu.

Cependant, des travaux futurs pourraient être réalisés pour assurer la *mise en ligne* des pairs et la synchronisation des séquences de pair-actions via, par exemple, un réseau P2P (à l'instar des blockchains). Le consensus pour s'assurer de l'ordre des actions pourrait être basé sur la valeur des condensats. Si deux pair-actions sont insérées par deux pairs différents à partir de la même pair-action parente, alors celle possédant la valeur de condensat la plus élevée pourrait être choisie comme réel successeur (la même méthode est utilisée par les réseaux de blockchains dans le cas où deux blocs sont minés en même temps). La pair-action concurrente devrait alors être recalculée et insérée à nouveau pour être prise en compte. Ceci permettrait

---

Avec les séquences de pair-actions, il nous semble particulièrement pertinent de pouvoir laisser le choix des artefacts à traiter aux pairs, selon leurs besoins.

donc de réaliser des actions simultanées sur un même artefact, sans trop alourdir le mécanisme des séquences de pair-actions.

6.4.8 SÉCURITÉ DE L'HISTORIQUE

La sécurité de la blockchain [61] est assurée en deux phases : la validation de blocs, et leur vérification. La validation permet d'ajouter un nouveau bloc à la blockchain par un consensus sur lequel tous les nœuds sont accordés. Ces consensus sont conçus de façon à ce qu'en pratique un attaquant ne puisse imposer sa propre version de la vérité sans une dépense inconcevable d'argent. La vérification, elle, est réalisée par tous les nœuds et permet de s'assurer le bloc reçu respecte à la fois le consensus établi et les blocs précédents. C'est pour cette raison que chaque nœud *doit* conserver une copie de la blockchain : il peut ainsi vérifier le respect du chaînage des blocs et la validité des transactions, sans risquer d'être trompé par un attaquant qui aurait modifié l'historique de la blockchain, entièrement ou partiellement. De fait, plus la blockchain est longue, et plus il est difficile pour un attaquant de modifier un bloc en son sein, puisque cela nécessiterait de recalculer tous les blocs suivants. On peut donc dire que la sécurité de la blockchain repose en partie sur le consensus et sur la conservation d'une copie locale par chaque nœud.

De façon assez similaire, la sécurité des séquences de pair-actions est elle aussi assurée par le fait qu'une copie locale de la séquence est gardée par les pairs concernés par l'artefact. De ce fait, toute modification de pair-actions *a posteriori* serait détectée par les pairs et l'historique pourrait être restaurée à sa dernière insertion de pair-action légitime. C'est le cas, par exemple, des pairs *stateful* (vu en Section 5.4) qui peuvent garder une trace de l'endroit où ils avaient arrêté leur dernière vérification dans la séquence même. Il existe néanmoins un

61. On parle ici de la construction de l'historique en lui-même, et non des transactions individuelles.

risque, où un nouvel intervenant sur l'artefact (qui n'a donc pas de copie locale antérieure de la séquence) reçoive une séquence tronquée par un attaquant. L'absence des k-dernières pair-actions n'est en effet pas détectable par un nouveau pair, puisque, pour une séquence de taille n, les $n-k$ premières transactions seront toujours valides si les k dernières sont enlevées. Dans le même scénario, il serait également possible de remplacer complètement l'historique par une séquence d'un artefact distinct mais partageant la même structure interne. Le seul moyen de détecter la fraude dans ces deux cas serait de comparer le résultat des modifications de la séquence avec l'artefact effectif. Des résultats différents permettrait de mettre en évidence un historique corrompu, mais ceci n'est pas toujours possible : *p. ex.*, les actions d'observations de la Section 5.1.2 n'ont pas d'effets visibles sur le contenu du document, ou bien l'attaquant pourrait avoir modifié la séquence de manière à dissimuler des actions intermédiaires sans que le contenu final diffère du document original.

La recherche de solutions à ces failles pourraient constituer le sujet de futurs travaux. Cependant, nous partageons une réflexion préliminaire sur une architecture visant à résoudre ces problèmes. Celle-ci pourrait s'articuler en deux temps. Premièrement, à chaque artefact pourrait être attribué un identifiant unique qui serait pris en compte dans le calcul du condensat de chaque pair-action. De cette façon, un historique ne pourrait être remplacé, même partiellement, par un autre puisque les condensats du faux historique ne correspondrait pas à l'identifiant de l'artefact original. Ceci nécessite néanmoins qu'un système d'attribution d'identifiants uniques soit mis en place et que ceux-ci ne puissent être altérés par un attaquant. Ce système pourrait être intégré directement dans la seconde phase de la résolution, à savoir la « mise en ligne » des pairs et la synchronisation des artefacts. Comme nous l'avons mentionné en Section 6.4.8, un réseau P2P pourrait être mis en place pour assurer la communication des artefacts et leurs séquences de pair-actions de façon automatisée, avec le moins de délai possible. Un artefact serait référencé par son identifiant unique au sein du réseau, et si une

tentative de troncature de l'historique a lieu, alors tous les pairs du réseau pourraient détecter l'attaque en tentant de se synchroniser à la nouvelle version de la séquence, qui ne correspondrait alors pas à l'ancienne version stockée en local. Les nouveaux intervenants sur un artefact seraient alors notifiés de cette tentative et pourraient choisir la « bonne » version de l'artefact (*p. ex.*, si une majorité de pairs valide la même).

6.4.9 APPLICABILITÉ

La question restante, et peut-être la plus importante, est le problème de l'applicabilité de ces approches dans des scénarios concrets. Contrairement au réseau Ethereum, aucune expérience à grande échelle incluant les séquences de pair-actions dans une situation réelle n'a été réalisée pour le moment. Cependant, les résultats et les comparaisons détaillés plus tôt nous permettent de tirer quelques conclusions.

La plus grande différence entre les blockchains et les séquences de pair-actions réside dans leur philosophie de gestion des historiques. Les premières adoptent un registre unique partagé auquel tous les utilisateurs participent, tandis que les secondes sont axées sur leur artefact : une séquence de pair-actions est associée à un unique artefact qui est lui-même manipulé par un sous-ensemble d'utilisateurs concernés. C'est pour cette raison qu'il est complexe de comparer strictement les performances et autres indicateurs de ces ceux approches. En effet, elles visent des cas d'application assez différents.

La blockchain est clairement conçue pour des contextes où des milliers de personnes interagissent simultanément entre elles. Ceci est principalement rendu possible par le fait qu'un utilisateur peut rejoindre le réseau très facilement, il lui suffit alors de se créer lui-même son porte-clés Ethereum. En intégrant dans son mécanisme même des frais de transaction, elle est particulièrement adaptée au contexte des transactions bancaires. Depuis l'arrivée des

smart contracts, elle est néanmoins envisagée dans de nombreux domaines pour remplacer les systèmes, souvent centralisés, mis en place jusqu'à présent. Les performances actuelles des réseaux, limitées en partie par les consensus, forcent cependant ces interactions à rester ponctuelles, et celles-ci ne peuvent profiter d'un traitement en temps réel à cause des délais requis par la validation d'un nouveau bloc. De ce fait, la faible scalabilité actuelle de la blockchain et les lourdeurs qu'elle impose (*p. ex.* reconception complète du système déjà présent, frais de transaction, consensus énergivores, nécessité d'être connecté à un réseau) la rendent difficilement applicable dans certains domaines.

Notre exemple sur la logistique hyperconnectée et ses ambitions montre déjà les limites d'un tel système, qui devrait permettre des taux de TPS bien plus élevés et proposer une solution efficace de stockage des transactions pour assurer un fonctionnement optimal. Aussi, nécessiter des frais de transactions de la part des employés d'une entreprise pour effectuer leur travail peut sembler quelque peu absurde, même si de l'argent factice peut être utilisé [62]. Ceci est valable également pour le *gas price* et la *gas limit* qui, dans le cadre de processus opérationnels, ne devrait pas contraindre l'exécution d'un *smart contract* développé pour l'occasion.

Pour ces raisons, il nous semblait important de conceptualiser une autre approche, la séquence de pair-action, qui est plus légère et reste décentralisée à la fois dans sa sécurisation et sa gestion des artefacts. Même si, dans leur formalise actuel, les séquences de pair-actions ne permettent pas de traiter des actions concurrentes, contrairement à la blockchain, elles sont bien plus rapides que cette dernière. Comme nous l'avons vu, ajouter une action allant jusqu'à 1 Ko avec Go-Artichoke prend seulement 0,12 ms, et environ 200 ms pour vérifier une séquence de 1 000 actions. Ceci signifie qu'un cycle complet de vérification d'une séquence

62. On peut en effet attribuer dès la mise en place du réseau une somme incommensurable d'Ether aux utilisateurs pour qu'ils ne soient jamais à court.

de 1 000 actions et ajout d'une nouvelle action prendrait moins d'un quart de seconde. Ces résultats nous semblent tout à fait adaptés pour une grande partie des situations décrites dans le Chapitre 2. Manifestement, dans l'exemple de formulaire médical, ajouter un quart de seconde à l'ouverture d'un fichier PDF est probablement parfaitement acceptable ; de plus, la validation pourrait même être effectuée en arrière-plan, rendant le document utilisable immédiatement pendant que sa validité est vérifiée en parallèle. Dans l'exemple de la carte de métro, ce délai de contrôle supplémentaire au tourniquet est aussi acceptable, étant donné le temps nécessaire à un passager pour simplement traverser le dispositif. Dans cette situation, intégrer un client blockchain dans un portillon de métro nous semblerait à la fois exagéré (à cause de la lourdeur du système à mettre en place) et contre-productif (la vérification des séquences de pair-actions serait facilement implémentable dans le logiciel interne du portillon, et son exécution plus rapide qu'avec la blockchain).

Le montant des informations devant être stockées dans la séquence de pair-actions est aussi raisonnable. Notre schéma, assez peu performant, de 440 octets par action [63] (ce qui est déjà probablement convenable pour des fichiers de type PDF) pourrait être réduit en utilisant une mise en forme des données plus optimale. Ceci implique néanmoins que 8 Ko (la taille d'une grande carte MIFARE) pourraient contenir à peu près 18 actions, ce qui est certainement assez, dans le cas de la carte de métro, pour suivre les déplacements d'un passager pendant une journée entière, en supposant que l'historique de la carte puisse être supprimé sans encombre à chaque nouveau jour.

Cependant, les deux approches se rejoignent complètement lorsqu'il s'agit de spécifier un cycle de vie et de vérifier son exécution. La blockchain et la séquence de pair-actions peuvent tout à fait servir de source d'événements à n'importe quel type de moniteur, à partir du moment où celui-ci est capable de communiquer avec la blockchain ou de lire des

63. Pour des actions de 100 octets.

données textuelles. Les propriétés du cycle de vie agissent alors comme de réelles boîtes noires complètement indépendantes de la façon dont sont implémentés les historiques. La seule différence notable sur ce point réside dans le fait qu'il est éventuellement possible d'intégrer directement la spécification des propriétés et leur monitoring dans la blockchain Ethereum à l'aide de *smart contracts*, là où seule la spécification, contenue dans une pair-action, est possible avec les séquences de pair-actions.

CONCLUSION

Par l'informatisation croissante de la société et de l'industrie, le domaine de la gestion de processus d'affaires (BPM) est en pleine expansion. En effet, celui-ci permet l'exécution efficace et rigoureuse de processus opérationnels, dans lesquels des multitudes de tâches sont à réaliser par, potentiellement, tout autant d'entités. Ces caractéristiques sont fortement liées à un des volets du BPM, le contrôle de conformité, qui assure que les exécutions de processus respectent les modèles et contraintes établis au préalable. L'avènement des processus opérationnels centrés sur les artefacts, introduits par Nigam & Caswell (2003), a entraîné une refonte du BPM par la définition des processus comme des ensembles d'opérations devant respecter les cycles de vie des artefacts en jeu à tout instant.

Un cycle de vie est constitué de l'ensemble des contraintes que les manipulations effectuées sur les artefacts doivent respecter. La spécification de cycles de vie est le sujet de nombreux travaux et recherches depuis les débuts de l'informatique et a beaucoup évolué. Un de leurs premiers représentants est la machine à états finis, qui permet de modéliser les séquences valides d'événements soumises à un système ou générées par ce dernier. Dans la même lignée, des formalismes de logiques temporelles, comme la logique temporelle linéaire (LTL) ont été conçus pour, là aussi, exprimer des contraintes sur le séquencement des événements. De nos jours, il existe un très grand nombre de spécifications possibles ; certaines, comme UML, sont même devenus des standards dans la planification et la conception de systèmes complexes, notamment logiciels. Selon les spécifications, les contraintes ainsi exprimées peuvent concerner l'ordonnancement des manipulations, leurs effets sur l'état interne des artefacts, ou encore les politiques de contrôle d'accès à appliquer.

Les travaux traitant du contrôle de conformité des processus par rapport à un cycle de vie sont tout aussi nombreux. Celui-ci consiste à s'assurer qu'un processus ou un système

produit un comportement respectueux de ses contraintes de cycle de vie. Pour accomplir cet objectif, les solutions se basent principalement sur deux types de vérification : le model checking et le runtime verification. Le premier consiste à concevoir un modèle abstrait du processus et vérifier que celui-ci respecte son cycle de vie ; de fait, si le modèle établi est valide, toutes les exécutions du système seront forcément conformes au cycle de vie. Le runtime verification propose une philosophie diamétralement opposée ; ici, le but est d'exécuter le système, qui produit alors des événements, et d'observer ces traces d'événements avec des moniteurs. Ces derniers sont alors responsables d'évaluer la conformité des comportements produits par rapport à ceux dictés par le cycle de vie. Si les travaux consacrés à ces approches sont nombreux, ceux proposant des solutions totalement décentralisées sont plus rares. La décentralisation de ces solutions peut être, en effet, souhaitable pour assurer des vérifications flexibles et qui ne se reposent pas sur une confiance préalable entre plusieurs pairs d'un processus. Cette thèse avait donc pour objectif de proposer des approches crédibles de contrôle décentralisé de conformité de processus opérationnels.

La première contribution de cette thèse est de proposer une architecture où l'ensemble des manipulations réalisées sur les artefacts est effectué par l'intermédiaire d'une blockchain. En l'occurrence, nous avons utilisé la plateforme Ethereum et implémenté des *smart contracts* pour stocker les actions provenant de divers acteurs ; ces derniers sont en réalité les agents d'une simulation AnyLogic, axée sur la livraison de colis dans un contexte de logistique urbaine hyperconnectée. La blockchain fournissant un registre partagé et sécurisé de transactions, chacun des acteurs est individuellement assuré de l'authenticité des manipulations qu'elle contient. Cet historique peut donc être utilisé comme une trace d'événements adaptée pour le runtime verification. Ainsi, chaque acteur peut effectuer la vérification de son côté, sans une quelconque confiance préalable envers ses pairs. En outre, nous avons énoncé plusieurs propriétés dont la vérification est pertinente dans le cadre de la supply chain ; elles concernent

le cycle de vie des colis à livrer, mais permettent également l'extraction de données statistiques destinées à l'évaluation des performances de la chaîne d'approvisionnement. Les moniteurs de ces propriétés ont ensuite été implémentés à l'aide de BeepBeep, une bibliothèque Java open source de Complex Event Processing. En outre, les expériences réalisées montrent que les moniteurs implémentés peuvent tout à fait traiter, en temps réel, les événements produits par une chaîne logistique de taille réelle.

Si les moniteurs sont capables d'effectuer une vérification en temps réel, l'approche par blockchain est limitée par certains de ses propres aspects techniques fondamentaux. Ceux-ci concernent les taux de transactions par seconde (TPS) atteignables, mais également l'immuabilité du registre, qui rend impossible la suppression de données superflues à long terme. Nous avons donc introduit une approche alternative basée sur les séquences de pair-actions, dont nous avons démontré l'intérêt dans un scénario de gestion de documents médicaux. Les séquences de pair-actions reprennent certains principes des blockchains, comme le chaînage des actions par leur condensat ; la confidentialité ainsi que l'authenticité de ces dernières sont assurées par des mécanismes de chiffrement et de signature. Néanmoins, elles se distinguent largement des blockchains traditionnelles par le fait qu'elles ne se reposent pas sur un consensus pour ajouter des actions, un facteur *très* limitant des blockchains. De plus, les historiques concernent des artefacts spécifiques, potentiellement pris individuellement, ce qui permet de cibler le traitement et le stockage des historiques par rapport à un regroupement pertinent d'artefacts, défini selon le contexte ; à l'opposé, la blockchain concentre toutes les actions d'un réseau, qui peuvent être tout à fait superflues pour certains acteurs. Les formalisations des séquences de pair-actions et leurs implémentations avec les langages Java et Go ont montré qu'elles étaient globalement plus rapides que les réseaux blockchains, avec un surplus en mémoire moins important, tout en proposant une flexibilité d'implémentation accrue par la

possibilité de choisir les formats de données ou les mécanismes de chiffrement/signature adaptés aux contextes d'utilisation.

Clairement, les deux approches permettent toutes deux l'adoption d'une vérification décentralisée du respect de cycles de vie d'artefacts. Cependant, leurs applications sont propices à des contextes d'utilisation différents. Comme nous l'avons montré, les blockchains sont plus adaptées à des scénarios où de multiples manipulations ont lieu en simultané sur, potentiellement, des artefacts identiques ; ceci est possible au coût d'un amoindrissement des performances et de la flexibilité sur le traitement et le stockage des transactions. Si les TPS des réseaux utilisés le permettent, il serait cependant intéressant d'envisager la mise en place de moniteurs, directement au sein de la blockchain. Dans ce cadre, une des contributions futures possibles serait de déployer des moniteurs sous la forme de *smart contracts* ; leurs méthodes seraient alors appelées par les *smart contracts* gérant le traitement des manipulations sur les artefacts. Ainsi, la vérification aurait lieu en même temps que l'ajout d'action, ce qui permettrait d'uniformiser son implémentation et, éventuellement, de pouvoir appliquer le runtime enforcement à la séquence de manipulations.

D'un autre côté, la formalisation et l'implémentation actuelles des séquences de pair-actions, bien que plus rapides et flexibles, ne permettent pas le traitement d'actions concurrentes : les manipulations sur un même artefact doivent être réalisées de manière séquentielle. Nous pensons que cette restriction pourrait être levée par l'adoption d'une architecture pair-à-pair (P2P) comme moyen de communication des pair-actions. Ainsi, les pairs pourraient se synchroniser immédiatement sur la dernière version valide d'une séquence, éliminant les risques de conflits. Cette approche permettrait également d'écarter les risques de troncature des historiques, présents dans l'implémentation actuelle ; cet aspect nous semble être également un sujet pertinent de travaux futurs.

À l'heure actuelle, il va de soi que les séquences de pair-actions ne sauraient rivaliser avec la popularité des réseaux de blockchain et de leurs écosystèmes. Ce sentiment est renforcé par le fait que la blockchain est encore une technologie jeune, émergente, améliorée continuellement, et poussée par de fortes communautés, notamment celle du logiciel libre. En outre, les annonces de nouvelles versions de certaines plateformes, comme Ethereum 2.0, laissent espérer l'obtention de TPS bien plus élevés et d'une fragmentation des registres, éliminant ainsi une part majeure des inconvénients des blockchains actuelles. Néanmoins, nous pensons que les séquences de pair-actions exposent une applicabilité prometteuse lorsqu'utilisées dans des scénarios plus modestes que ceux des blockchains ; cette supposition est motivée par leur simplicité d'utilisation, leur flexibilité et leurs performances, globalement suffisantes pour bon nombre de contextes d'application, tels que la gestion de documents médicaux ou l'accès aux transports en commun par carte à puce.

BIBLIOGRAPHIE

Abeyratne, S. A. & Monfared, R. P. (2016). Blockchain ready manufacturing supply chain using distributed ledger. *International Journal of Research in Engineering and Technology*, *5*(9), 1–10. doi: 10.15623/ijret.2016.0509001

Adi, A., Botzer, D., Nechushtai, G. & Sharon, G. (2006). Complex event processing for financial services. Dans *Proceedings of the 2006 IEEE Services Computing Workshops (SCW 2006), Chicago, IL*, pp. 7–12. doi: 10.1109/SCW.2006.7

Agbo, C. C., Mahmoud, Q. H. & Eklund, J. M. (2019). Blockchain technology in healthcare : A systematic review. *Healthcare*, *7*(2), 56. doi: 10.3390/healthcare7020056

Alvarez, J. (2019). *Who is Satoshi Nakamoto ? A look at the candidates*. Repéré le 30 août 2019, à https://blockonomi.com/who-is-satoshi-nakamoto/

Angelis, S. D., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A. & Sassone, V. (2018). PBFT vs proof-of-authority : Applying the CAP theorem to permissioned blockchain. Dans E. Ferrari, M. Baldi, & R. Baldoni (Éds.). *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy*. doi: 10.5281/zenodo.1169273

Ataullah, A. A. & Tompa, F. W. (2011). Business policy modeling and enforcement in databases. *Proceedings of the VLDB Endowment*, *4*(11), 921–931. doi: 10.14778/3402707.3402730

Ateniese, G., Steiner, M. & Tsudik, G. (2000). New multiparty authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communications*, *18*(4), 628–639. doi: 10.1109/49.839937

Azaria, A., Ekblaw, A., Vieira, T. & Lippman, A. (2016). MedRec : Using blockchain for medical data access and permission management. Dans I. Awan & M. Younas (Éds.). *2nd International Conference on Open and Big Data, OBD 2016, Vienna, Austria*, pp. 25–30. doi: 10.1109/OBD.2016.11

Back, A. (2002). *Hashcash - A denial of service counter-measure*. Repéré le 30 août 2019, à http://www.hashcash.org/papers/hashcash.pdf

Baran, P. (1964). On distributed communications networks. *IEEE Transactions on Communications Systems*, *12*(1), 1–9. doi: 10.1109/TCOM.1964.1088883

Barker, E. (2016). Recommendation for key management – Part 1 : General. *NIST SP 800-57pt1r5*. doi: 10.6028/NIST.SP.800-57pt1r5

Barker, E., Johnson, D. & Smid, M. (2006). Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. *NIST SP 800-56A*. doi: 10.6028/NIST.SP.800-56A

Bartocci, E., Falcone, Y., Francalanza, A. & Reger, G. (2018). Introduction to runtime verification. Dans E. Bartocci & Y. Falcone (Éds.). *Lectures on Runtime Verification - Introductory and Advanced Topics* (pp. 1–33). doi: 10.1007/978-3-319-75632-5_1

Bauer, A. & Falcone, Y. (2016). Decentralised LTL monitoring. *Formal Methods in System Design*, *48*(1-2), 46–93. doi: 10.1007/s10703-016-0253-8

Bauer, A. K. & Falcone, Y. (2012). Decentralised LTL monitoring. Dans D. Giannakopoulou & D. Méry (Éds.). *FM 2012 : Formal Methods - 18th International Symposium, Paris, France*, pp. 85–100. doi: 10.1007/978-3-642-32759-9_10

Baumann, H., Grassle, P. & Baumann, P. (2005). *UML 2.0 in action : A project-based tutorial*. Repéré à https://www.packtpub.com/product/uml-2-0-in-action-a-project-based-tutorial/9781904811558

Beall, A. (2017). *Bitcoin mining uses more energy than Ecuador – But there's a fix*. Repéré le 30 août 2019, à https://www.newscientist.com/article/2151823-bitcoin-mining-uses-more-energy-than-ecuador-but-theres-a-fix/

Benchmarks Game (2020). *Go vs Java - Which programs are fastest ?* Repéré le 07 août 2020, à https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go.html

Berry, A. & Milosevic, Z. (2013). Real-time analytics for legacy data streams in health : Monitoring health data quality. Dans D. Gasevic, M. Hatala, H. R. M. Nezhad, & M. Reichert (Éds.). *17th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2013), Vancouver, BC*, pp. 91–100. doi: 10.1109/EDOC.2013.19

Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Van Assche, G. & Van Keer, R. (2020). *Software performance figures*. Repéré le 07 août 2020, à https://keccak.team/sw_performance.html

Betti, Q. & Hallé, S. (2019). *Benchmark for supply chain monitoring properties using BeepBeep (LabPal dataset)*. doi: 10.5281/zenodo.3066013

Betti, Q., Khoury, R., Hallé, S. & Montreuil, B. (2019). Improving hyperconnected logistics with blockchains and smart contracts. *IT Professional*, *21*(4), 25–32. doi: 10.1109/MITP.2019.2912135

Betti, Q., Montreuil, B., Khoury, R. & Hallé, S. (2020). Smart contracts-enabled simulation for hyperconnected logistics. Dans M. A. Khan, M. T. Quasim, F. Algarni, & A. Alharthi (Éds.). *Decentralised Internet of Things : A Blockchain Perspective* (pp. 109–149). doi: 10.1007/978-3-030-38677-1_6

Bhattacharya, K., Gerede, C. E., Hull, R., Liu, R. & Su, J. (2007). Towards formal analysis of artifact-centric business process models. Dans G. Alonso, P. Dadam, & M. Rosemann (Éds.). *Business Process Management, 5th International Conference (BPM 2007), Brisbane, Australia*, pp. 288–304. doi: 10.1007/978-3-540-75183-0_21

Biba, K. J. (1977). *Integrity considerations for secure computer systems*. Repéré le 19 septembre 2020, à https://www.researchgate.net/publication/235043659_Integrity_Considerations_for_Secure_Computer_Systems

Bielova, N. & Massacci, F. (2011). Predictability of enforcement. Dans Ú. Erlingsson, R. J. Wieringa, & N. Zannone (Éds.). *Engineering Secure Software and Systems - Third International Symposium (ESSoS 2011), Madrid, Spain*, Vol. 6542, pp. 73–86. doi: 10.1007/978-3-642-19125-1_6

Bielova, N., Massacci, F. & Micheletti, A. (2009). Towards practical enforcement theories. Dans A. Jøsang, T. Maseng, & S. J. Knapskog (Éds.). *Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems (NordSec 2009), Oslo, Norway*, pp. 239–254. doi: 10.1007/978-3-642-04766-4_17

BigchainDB GmbH (2018). *BigchainDB 2.0 - The blockchain database*. Repéré le 27 juillet 2019, à https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf

BitPay (2020). *Welcome to the future of payments*. Repéré le 29 juillet 2020, à https://bitpay.com/

Bitshares (2019). *Delegated proof-of-stake consensus*. Repéré le 20 juillet 2019, à https://bitshares.org/technology/delegated-proof-of-stake-consensus/

Blockchair (2020). *Ethereum blockchain size chart*. Repéré le 15 août 2020, à https://blockchair.com/ethereum/charts/blockchain-size

Block.one (2018). *EOS.IO technical white paper v2*. Repéré le 20 juillet 2019, à https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

Boebert, W. & Kain, R. (1985). A practical alternative to hierarchical integrity policies. Dans *Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD*, pp. 18–27. Repéré à https://csrc.nist.rip/publications/history/nissc/1985-8th-NCSC-proceedings.pdf

Boussaha, M. R., Khoury, R. & Hallé, S. (2017). Monitoring of security properties using BeepBeep. Dans A. Imine, J. M. Fernandez, J. Marion, L. Logrippo, & J. García-Alfaro (Éds.). *Foundations and Practice of Security - 10th International Symposium (FPS 2017), Nancy, France*, pp. 160–169. doi: 10.1007/978-3-319-75650-9_11

Breitner, J. & Heninger, N. (2019). *Biased nonce sense : Lattice attacks against weak ECDSA signatures in cryptocurrencies*. Repéré le 13 février 2020, à https://eprint.iacr.org/2019/023

Brewer, D. F. C. & Nash, M. J. (1989). The Chinese wall security policy. Dans *Proceedings of the 1989 IEEE Symposium on Security and Privacy, Oakland, CA*, pp. 206–214. doi: 10.1109/SECPRI.1989.36295

Brown, D. R. L. (2009). *Standards for efficient cryptography 1 (SEC 1)*. Repéré le 13 février 2020, à http://secg.org/sec1-v2.pdf

Brown, D. R. L. (2010). *Standards for efficient cryptography 2 (SEC 2)*. Repéré le 07 août 2020, à http://www.secg.org/sec2-v2.pdf

BTC.com (2019). *Pool stats*. Repéré le 30 août 2019, à https://btc.com/stats/pool

Buterin, V. (2016). *Pyethereum source code*. Repéré le 27 août 2019, à https://github.com/ethereum/pyethereum

Calvanese, D., De Giacomo, G., Hull, R. & Su, J. (2009). Artifact-centric workflow dominance. Dans L. Baresi, C. Chi, & J. Suzuki (Éds.). *Service-Oriented Computing, 7th International Joint Conference (ICSOC-ServiceWave 2009), Stockholm, Sweden*, pp. 130–143. doi: 10.1007/978-3-642-10383-4_9

Calvar, J., Tremblay-Lessard, R. & Hallé, S. (2012). A runtime monitoring framework for event streams with non-primitive arguments. Dans G. Antoniol, A. Bertolino, & Y. Labiche (Éds.). *Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST 2012), Montreal, QC*, pp. 499–508. doi: 10.1109/ICST.2012.135

Canellis, D. (2018). *We are thinking about Bitcoin's energy usage in all the wrong ways*. Repéré le 30 août 2019, à https://thenextweb.com/hardfork/2018/08/28/bitcoin-drives-energy-innovation/

Canellis, D. (2019). *Bitcoin has Nearly 100,000 nodes, but over 50% run vulnerable code*. Repéré le 30 mai 2020, à https://thenextweb.com/hardfork/2019/05/06/bitcoin-100000-nodes-vulnerable-cryptocurrency/

Christopher, M. (2016). *Logistics and supply chain management* (5th ed.). London, United Kingdom: Pearson.

Clarke, E. M. & Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. Dans D. Kozen (Éd.). *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, Vol. 131, pp. 52–71. doi: 10.1007/BFb0025774

Clarke, E. M., Henzinger, T. A. & Veith, H. (2018). Introduction to model checking. Dans E. M. Clarke, T. A. Henzinger, H. Veith, & R. Bloem (Éds.). *Handbook of Model Checking* (pp. 1–26). doi: 10.1007/978-3-319-10575-8_1

Coinbase Commerce (2020). *Accept Bitcoin payments within minutes*. Repéré le 29 juillet

2020, à https://commerce.coinbase.com/

CoinGate (2020). *Point-of-Sale (PoS) app for Bitcoin & other coins*. Repéré le 29 juillet 2020, à https://coingate.com/pos

CoinMarketCap (2019). *Cryptocurrency market capitalizations*. Repéré le 30 août 2019, à https://coinmarketcap.com/

Cook, M. R. (2015). *Go vs Java performance comparison*. Repéré le 07 août 2020, à https://mrcook.uk/golang-vs-java-performance

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W. T. *et al.* (2008). Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC 5280*

Coppock, M. (2017). *Bitcoin and Ethereum cryptocurrency mining put a real burden on the electrical grid*. Repéré le 30 août 2019, à https://www.digitaltrends.com/computing/bitcoin-ethereum-mining-use-significant-electrical-power/

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). Repéré à https://mitpress.mit.edu/books/introduction-algorithms-third-edition

Crainic, T. G. & Montreuil, B. (2016). Physical Internet enabled hyperconnected city logistics. *Transportation Research Procedia*, *12*, 383–398. doi: 10.1016/j.trpro.2016.02.074

CryptoLions (2019). *EOS network monitor*. Repéré le 20 juillet 2019, à https://eosnetworkmonitor.io/

Dai, W. (2009). *Speed comparison of popular crypto algorithms*. Repéré le 08 juillet 2020, à https://cryptopp.com/benchmarks.html

Dang, Q. H. (2015). Secure Hash Standard. *NIST FIPS 180-4*. doi: 10.6028/NIST.FIPS.180-4

Datanyze (2020). *Business process management market share report*. Repéré le 29 octobre

2020, à https://www.datanyze.com/market-share/business-process-management--452

Debian (2019). *Debian manpages — crypt(3)*. Repéré le 28 avril 2020, à https://manpages.debian.org/buster/manpages-dev/crypt.3.en.html

Deetman, S. (2016). *Bitcoin could consume as much electricity as Denmark by 2020*. Repéré le 30 août 2019, à https://www.vice.com/en_us/article/aek3za/bitcoin-could-consume-as-much-electricity-as-denmark-by-2020

Desel, J. & Juhás, G. (2001). "What is a Petri net ?" Informal answers for the informed reader. Dans H. Ehrig, J. Padberg, G. Juhás, & G. Rozenberg (Éds.). *Unifying Petri Nets : Advances in Petri Nets*, Vol. 2128, pp. 1–25. doi: 10.1007/3-540-45541-8_1

Diffie, W. & Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), 644–654. doi: 10.1109/TIT.1976.1055638

Domingo, C. (2017). *The Bitcoin vs Visa electricity consumption fallacy*. Repéré le 30 août 2019, à https://hackernoon.com/the-bitcoin-vs-visa-electricity-consumption-fallacy-8cf194987a50

Dorri, A., Kanhere, S. S., Jurdak, R. & Gauravaram, P. (2017). Blockchain for IoT security and privacy : The case study of a smart home. Dans *2017 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2017, Kona, HI*, pp. 618–623. doi: 10.1109/PERCOMW.2017.7917634

Dorri, A., Steger, M., Kanhere, S. S. & Jurdak, R. (2017). Blockchain : A distributed solution to automotive security and privacy. *IEEE Communications Magazine*, *55*(12), 119–125. doi: 10.1109/MCOM.2017.1700879

Dumas, M., Rosa, M. L., Mendling, J. & Reijers, H. A. (2018). *Fundamentals of business process management* (2nd ed.). doi: 10.1007/978-3-662-56509-4

Dworkin, M. (2001). Recommendation for block cipher modes of operation : Methods and techniques. *NIST SP 800-38A*. doi: 10.6028/NIST.SP.800-38A

Dworkin, M. (2007). Recommendation for block cipher modes of operation : Galois/Counter Mode (GCM) and GMAC. *NIST SP 800-38D*. doi: 10.6028/NIST.SP.800-38D

ECMA International (2013). *The JSON data interchange format (ECMA-404), 1st Ed.* Repéré le 19 mai 2020, à https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-404%201st%20edition%20October%202013.pdf

Emerson, E. A. (1990). Temporal and modal logic. Dans J. van Leeuwen (Éd.). *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics* (pp. 995–1072). doi: 10.1016/b978-0-444-88074-1.50021-4

Emerson, E. A. & Halpern, J. Y. (1986). "Sometimes" and "not never" revisited : On branching versus linear time temporal logic. *Journal of the ACM*, *33*(1), 151–178. doi: 10.1145/4904.4999

Ethereum Foundation (2017). *Proof of stake FAQ*. Repéré le 12 septembre 2019, à https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ

Ethereum Foundation (2019). *Ethash*. Repéré le 30 août 2019, à https://github.com/ethereum/wiki/wiki/Ethash

Ethereum Foundation (2019). *Solidity 0.5.10 documentation*. Repéré le 20 juillet 2019, à https://solidity.readthedocs.io/en/v0.5.10/

Ethereum Foundation (2020). *Ethereum 2.0 (Eth2)*. Repéré le 15 août 2020, à https://ethereum.org/en/eth2/

Ethernodes (2020). *Clients*. Repéré le 30 juin 2020, à https://ethernodes.org/

Etherscan (2019). *Ethereum unique address growth chart*. Repéré le 02 juin 2020, à https://etherscan.io/chart/address

Falcone, Y. (2010). You should better enforce than verify. Dans H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, & N. Tillmann (Éds.). *Runtime Verification - First International Conference (RV 2010), St. Julians, Malta*, pp.

89–105. doi: 10.1007/978-3-642-16612-9_9

Falcone, Y., Cornebize, T. & Fernandez, J. (2014). Efficient and generalized decentralized monitoring of regular languages. Dans E. Ábrahám & C. Palamidessi (Éds.). *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference (FORTE 2014), Held as Part of the 9th International Federated Conference on Distributed Computing Techniques (DisCoTec 2014), Berlin, Germany*, pp. 66–83. doi: 10.1007/978-3-662-43613-4_5

Falcone, Y., Jéron, T., Marchand, H. & Pinisetty, S. (2016). Runtime enforcement of regular timed properties by suppressing and delaying events. *Systems & Control Letters*, *123*, 2–41. doi: 10.1016/j.scico.2016.02.008

Falcone, Y., Mounier, L., Fernandez, J. & Richier, J. (2011). Runtime enforcement monitors : Composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, *38*(3), 223–262. doi: 10.1007/s10703-011-0114-4

Falcone, Y. & Pinisetty, S. (2019). On the runtime enforcement of timed properties. Dans B. Finkbeiner & L. Mariani (Éds.). *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal*, Vol. 11757, pp. 48–69. doi: 10.1007/978-3-030-32079-9_4

Feng Tian (2017). A supply chain traceability system for food safety based on HACCP, blockchain & Internet of things. Dans *2017 International Conference on Service Systems and Service Management, Dalian, China*, pp. 1–6. doi: 10.1109/ICSSSM.2017.7996119

Ferraiolo, D. & Kuhn, D. (1992). Role-based access controls. Dans *Proceedings of the 15th National Computer Security Conference, Baltimore, MD*, p. 554–563.

Fielding, R. T. & Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, *2*(2), 115–150. doi: 10.1145/514183.514185

Frankenfield, J. (2019). *51% attack*. Repéré le 30 août 2019, à https://www.investopedia.com/terms/1/51-attack.asp

Futurepia (2019). *Futurepia : Built and owned by user, white paper ver. 1.3*. Repéré le 30 juillet 2019, à https://futurepia.io/assets/img/FUTUREPIA_WhitePaper_EN.pdf

Futurepia (2019). *Futurepia mainnet, blockchain for social media*. Repéré le 20 juillet 2019, à https://futurepia.io/

Gerede, C. E. & Su, J. (2007). Specification and verification of artifact behaviors in business process models. Dans B. J. Krämer, K. Lin, & P. Narasimhan (Éds.). *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria*, pp. 181–192. doi: 10.1007/978-3-540-74974-5_15

Giusto, D., Iera, A., Morabito, G. & Atzori, L. (Éds.) (2010). *The Internet of Things : 20th Tyrrhenian workshop on digital communications*. doi: 10.1007/978-1-4419-1674-7

Gonzalez, P., Griesmayer, A. & Lomuscio, A. (2012). Verifying GSM-based business artifacts. Dans C. A. Goble, P. P. Chen, & J. Zhang (Éds.). *2012 IEEE 19th International Conference on Web Services, Honolulu, HI*, pp. 25–32. doi: 10.1109/ICWS.2012.31

Gopalakrishnan, G. (2019). *Automata and computability : Programmer's perspective*. Repéré à https://www.routledge.com/Automata-and-Computability-A-Programmers-Perspective/Gopalakrishnan/p/book/9780367656546

Grand View Research (2016). *Business process management market size report, forecasts to 2024*. Repéré le 29 octobre 2020, à https://www.grandviewresearch.com/industry-analysis/business-process-management-bpm-market

Hallé, S. (2018). *Event stream processing with BeepBeep 3 : Log crunching and analysis made easy*. Repéré à https://www.puq.ca/catalogue/livres/event-stream-processing-with-beep-beep-3663.html

Hallé, S., Villemaire, R. & Cherkaoui, O. (2009). Specifying and validating data-aware temporal web service properties. *IEEE Transactions on Software Engineering*, *35*(5), 669–683. doi: 10.1109/TSE.2009.29

Hallé, S. (2013). Cooperative runtime monitoring. *Enterprise IS*, *7*(4), 395–423. doi: 10.1080/17517575.2012.721118

Hallé, S., Gaboury, S. & Bouchard, B. (2016). Activity recognition through complex event processing : First findings. Dans B. Bouchard, S. Giroux, A. Bouzouane, & S. Gaboury

(Éds.). *Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop, Phoenix, AZ*. Repéré à https://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12561

Hallé, S., Gaboury, S. & Bouchard, B. (2016). Towards user activity recognition through energy usage analysis and complex event processing. Dans *Proceedings of the 9th ACM International Conference on PErvasive Technologies Related to Assistive Environments (PETRA 2016), Corfu Island, Greece*, p. 3. doi: 10.1145/2910674

Hallé, S., Gaboury, S. & Khoury, R. (2016). A glue language for event stream processing. Dans J. Joshi, G. Karypis, L. Liu, X. Hu, R. Ak, Y. Xia, W. Xu, A. Sato, S. Rachuri, L. H. Ungar, P. S. Yu, R. Govindaraju, & T. Suzumura (Éds.). *2016 IEEE International Conference on Big Data (BigData 2016), Washington, DC*, pp. 2384–2391. doi: 10.1109/BigData.2016.7840873

Hallé, S. & Khoury, R. (2017). SealTest : A simple library for test sequence generation. Dans T. Bultan & K. Sen (Éds.). *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA*, pp. 392–395. doi: 10.1145/3092703.3098229

Hallé, S. & Khoury, R. (2018). Writing domain-specific languages for beepbeep. Dans C. Colombo & M. Leucker (Éds.). *Runtime Verification - 18th International Conference (RV 2018), Limassol, Cyprus*, pp. 447–457. doi: 10.1007/978-3-030-03769-7_27

Hallé, S., Khoury, R. & Awesso, M. (2018). Streamlining the inclusion of computer experiments in a research paper. *IEEE Computer*, *51*(11), 78–89. doi: 10.1109/MC.2018.2876075

Hallé, S., Khoury, R., Betti, Q., El-Hokayem, A. & Falcone, Y. (2018). Decentralized enforcement of document lifecycle constraints. *Information Systems*, *74*(Part), 117–135. doi: 10.1016/j.is.2017.08.002

Hallé, S. & Villemaire, R. (2012). Runtime enforcement of web service message contracts with data. *IEEE Trans. Services Computing*, *5*(2), 192–206. doi: 10.1109/TSC.2011.10

Harel, D. (1987). Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231–274. doi: 10.1016/0167-6423(87)90035-9

Harel, D. & Pnueli, A. (1985). On the development of reactive systems. Dans K. R. Apt (Éd.). *Logics and Models of Concurrent Systems*, Vol. 13, pp. 477–498. doi: 10.1007/978-3-642-82453-1_17

Hariri, B. B., Calvanese, D., De Giacomo, G., De Masellis, R. & Felli, P. (2011). Foundations of relational artifacts verification. Dans S. Rinderle-Ma, F. Toumani, & K. Wolf (Éds.). *Business Process Management - 9th International Conference (BPM 2011), Clermont-Ferrand, France*, pp. 379–395. doi: 10.1007/978-3-642-23059-2_28

Hölbl, M., Kompara, M., Kamisalic, A. & Zlatolas, L. N. (2018). A systematic review of the use of blockchain in healthcare. *Symmetry*, *10*(10), 470. doi: 10.3390/sym10100470

Hopcroft, J. E., Motwani, R. & Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation* (3rd ed.). Repéré à https://www.pearson.fr/fr/book/?GCOI=27440106765310

Hull, R., Damaggio, E., De Masellis, R., Fournier, F., Gupta, M., Heath, F. T., Hobson, S., Linehan, M. H., Maradugu, S., Nigam, A., Sukaviriya, P. N. & Vaculín, R. (2011). Business artifacts with guard-stage-milestone lifecycles : Managing artifact interactions with conditions and events. Dans D. M. Eyers, O. Etzion, A. Gal, S. B. Zdonik, & P. Vincent (Éds.). *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems (DEBS 2011), New York, NY*, pp. 51–62. doi: 10.1145/2002259.2002270

Infante, R. (2018). *Transaction lifecycle on the Ethereum blockchain*. Repéré le 28 août 2019, à https://medium.com/@roberto.g.infante/transaction-life-cycle-on-the-ethereum-blockchain-b0d92fa73fb1

Infante, R. (2019). *Building Ethereum Dapps : Decentralized applications on the Ethereum blockchain*. Repéré à https://www.manning.com/books/building-ethereum-dapps

Jia, X., Wenming, Y. & Dong, W. (2009). Complex event processing model for distributed RFID network. Dans S. Sohn, L. Chen, S. Hwang, K. Cho, S. Kawata, K. Um, F. I. S. Ko, K. Kwack, J. H. Lee, G. Kou, K. Nakamura, A. C. M. Fong, & P. C. M. Ma (Éds.). *Proceedings of the 2nd International Conference on Interaction Sciences : Information Technology, Culture and Human 2009, Seoul, Korea*, pp. 1219–1222. doi: 10.1145/1655925.1656147

Jin, D., Meredith, P. O., Lee, C. & Rosu, G. (2012). JavaMOP : Efficient parametric

runtime monitoring framework. Dans M. Glinz, G. C. Murphy, & M. Pezzè (Éds.). *34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland*, pp. 1427–1430. doi: 10.1109/ICSE.2012.6227231

Johansen, H. D., Birrell, E., van Renesse, R., Schneider, F. B., Stenhaug, M. & Johansen, D. (2015). Enforcing privacy policies with meta-code. Dans K. Kono & T. Shinagawa (Éds.). *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys 2015), Tokyo, Japan*, pp. 16 :1–16 :7. doi: 10.1145/2797022.2797040

Johnson, D., Menezes, A. & Vanstone, S. A. (2001). The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, *1*(1), 36–63. doi: 10.1007/s102070100002

Jonsson, J. & Kaliski, B. (2003). Public-key cryptography standards (PKCS) #1 : RSA cryptography specifications version 2.1. *RFC 3447*, (3447)

Josefsson, S. (2006). The base16, base32, and base64 data encodings. *RFC 4648*

Josefsson, S. & Percival, C. (2016). The scrypt password-based key derivation function. *RFC 7914*

Kaboudvand, S., Montreuil, B., Buckley, S. & Faugere, L. (2018). Hyperconnected megacity logistics service network assessment : A simulation sandbox approach [Keynote speech]. 2018 IISE Annual Conference, Orlando, FL.

Khan, M. A. & Salah, K. (2018). IoT security : Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, *82*, 395–411. doi: 10.1016/j.future.2017.11.022

Khoury, R., Hallé, S. & Waldmann, O. (2016). Execution trace analysis using LTL-FO+. Dans T. Margaria & B. Steffen (Éds.). *Leveraging Applications of Formal Methods, Verification and Validation : Discussion, Dissemination, Applications - 7th International Symposium (ISoLA 2016), Corfu, Greece*, pp. 356–362. doi: 10.1007/978-3-319-47169-3_26

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of Computation*, *48*(177),

203–209. doi: 10.1090/S0025-5718-1987-0866109-5

Kripke, S. (1959). A completeness theorem in modal logic. *The Journal of Symbolic Logic*, *24*(1), 1–14. doi: 10.2307/2964568

Kshetri, N. (2017). Can blockchain strengthen the Internet of Things ? *IT Professional*, *19*(4), 68–72. doi: 10.1109/MITP.2017.3051335

Kshetri, N. (2018). Blockchain's roles in meeting key supply chain management objectives. *International Journal of Information Management*, *39*, 80–89. doi: 10.1016/j.ijinfomgt.2017.12.005

Kumaran, S., Liu, R. & Wu, F. Y. (2008). On the duality of information-centric and activity-centric models of business processes. Dans Z. Bellahsene & M. Léonard (Éds.). *Advanced Information Systems Engineering, 20th International Conference (CAiSE 2008), Montpellier, France*, pp. 32–47. doi: 10.1007/978-3-540-69534-9_3

La, V. H., Fuentes-Samaniego, R. A. & Cavalli, A. R. (2016). Network monitoring using MMT : An application based on the user-agent field in HTTP headers. Dans L. Barolli, M. Takizawa, T. Enokido, A. J. Jara, & Y. Bocchi (Éds.). *30th IEEE International Conference on Advanced Information Networking and Applications (AINA 2016), Crans-Montana, Switzerland*, pp. 147–154. doi: 10.1109/AINA.2016.41

Lamport, L., Shostak, R. E. & Pease, M. C. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *4*(3), 382–401. doi: 10.1145/357172.357176

Lano, K. (2009). *UML 2 semantics and applications*. Repéré à https://www.wiley.com/en-us/UML+2+Semantics+and+Applications-p-9780470522615

Leucker, M. & Schallhart, C. (2009). A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, *78*(5), 293–303. doi: 10.1016/j.jlap.2008.08.004

Liang, X., Zhao, J., Shetty, S., Liu, J. & Li, D. (2017). Integrating blockchain for data sharing and collaboration in mobile healthcare applications. Dans *28th IEEE Annual International Symposium on Personal, Indoor, and Mobile Radio Communications, PIMRC*

2017, Montreal, QC, pp. 1–5. doi: 10.1109/PIMRC.2017.8292361

López-Pintado, O., García-Bañuelos, L., Dumas, M. & Weber, I. (2017). Caterpillar : A blockchain-based business process management system. Dans R. Clarisó, H. Leopold, J. Mendling, W. M. P. van der Aalst, A. Kumar, B. T. Pentland, & M. Weske (Éds.). *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain*, Vol. 1920 de *CEUR Workshop Proceedings*. Repéré à http://ceur-ws.org/Vol-1920/BPM_2017_paper_199.pdf

Luckham, D. (2008). The power of events : An introduction to complex event processing in distributed enterprise systems. Dans N. Bassiliades, G. Governatori, & A. Paschke (Éds.). *Rule Representation, Interchange and Reasoning on the Web (RuleML 2008)*. doi: 10.1007/978-3-540-88808-6_2

Madhwal, Y. & Panfilov, P. (2017). Industrial case : Blockchain on aircraft's parts supply chain management. Dans *American Conference on Information Systems 2017, Workshop on Smart Manufacturing Proceedings*. Repéré à https://aisel.aisnet.org/sigbd2017/6/

MarketsandMarkets (2020). *Business process management market size, share and global forecast to 2025*. Repéré le 29 octobre 2020, à https://www.marketsandmarkets.com/Market-Reports/business-process-management-market-157890056.html

McCook, H. (2014). *An order-of-magnitude estimate of the relative sustainability of the Bitcoin network, 2nd Ed.* Repéré le 30 août 2019, à https://www.academia.edu/10701244/Relative_Sustainability_-_2nd_Edition

Mealy, G. H. (1955). A method for synthesizing sequential circuits. *The Bell System Technical Journal*, *34*(5), 1045–1079. doi: 10.1002/j.1538-7305.1955.tb03788.x

Mendling, J., Weber, I., van der Aalst, W. M. P., vom Brocke, J., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C. D., Dumas, M., Dustdar, S., Gal, A., García-Bañuelos, L., Governatori, G., Hull, R., Rosa, M. L., Leopold, H., Leymann, F., Recker, J., Reichert, M., Reijers, H. A., Rinderle-Ma, S., Solti, A., Rosemann, M., Schulte, S., Singh, M. P., Slaats, T., Staples, M., Weber, B., Weidlich, M., Weske, M., Xu, X. & Zhu, L. (2018). Blockchains for business process management - Challenges and opportunities. *ACM Transactions on Management Information Systems*, *9*(1), 4 :1–4 :16. doi: 10.1145/3183367

Menezes, A. J., van Oorschot, P. C. & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. Discrete mathematics and its applications. Repéré à https://www.routledge.com/Handbook-of-Applied-Cryptography/Menezes-van-Oorschot-Vanstone/p/book/9780849385230

Merkle, R. C. (1982). *Method of providing digital signatures* [Brevet]. [Univ Leland Stanford Junior]. US4309569 (A). Brevet CIB : H04L9/32 ; (IPC1-7) : H04L9/00. Repéré le 31 août 2019, à https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=19820105&DB=&locale=en_EP&CC=US&NR=4309569A&KC=A&ND=2

Mettler, M. (2016). Blockchain technology in healthcare : The revolution starts here. Dans *18th IEEE International Conference on e-Health Networking, Applications and Services, Healthcom 2016, Munich, Germany*, pp. 1–3. doi: 10.1109/HealthCom.2016.7749510

Meyer, A., Pufahl, L., Fahland, D. & Weske, M. (2013). Modeling and enacting complex data dependencies in business processes. Dans F. Daniel, J. Wang, & B. Weber (Éds.). *Business Process Management - 11th International Conference (BPM 2013), Beijing, China*, pp. 171–186. doi: 10.1007/978-3-642-40176-3_14

Miller, V. S. (1986). Use of elliptic curves in cryptography. Dans H. C. Williams (Éd.). *Advances in Cryptology – CRYPTO '85 Proceedings, Santa Barbara, CA*, pp. 417–426. doi: 10.1007/3-540-39799-X_31

Mo, L. (2011). A study on modern agricultural products logistics supply chain management mode based on IoT. Dans *Second International Conference on Digital Manufacturing and Automation (ICDMA 2011), Zhangjiajie, Hunan*, pp. 117–120. doi: 10.1109/ICDMA.2011.36

Montreuil, B. (2011). Toward a Physical Internet : Meeting the global logistics sustainability grand challenge. *Logistics Research*, *3*(2-3), 71–87. doi: 10.1007/s12159-011-0045-x

Montreuil, B., Buckley, S., Faugere, L., Khir, R. & Derhami, S. (2018). Urban parcel logistics hub and network design : The impact of modularity and hyperconnectivity. Dans *15th International Material Handling Research Colloquium (IMHRC) Proceedings, Savannah, GA*, Vol. 19. Repéré à https://digitalcommons.georgiasouthern.edu/pmhr_2018/19/

Moore, E. F. (1956). Gedanken-experiments on sequential machines. *Automata Studies*, *34*.

doi: 10.1515/9781400882618-006

Morrison, D. R. (1968). PATRICIA - Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, *15*(4), 514–534. doi: 10.1145/321479.321481

Mukkamala, R. R., Hildebrandt, T. T. & Tøth, J. B. (2008). The resultmaker online consultant : From declarative workflow management in practice to LTL. Dans M. van Sinderen, J. P. A. Almeida, L. F. Pires, & M. Steen (Éds.). *2008 12th Enterprise Distributed Object Computing Conference Workshops, Munich, Germany*, pp. 135–142. doi: 10.1109/EDOCW.2008.57

Munoz, D., Constantinescu, D., Asenjo, R. & Fuentes, L. (2019). ClinicAppChain : A low-cost blockchain hyperledger solution for healthcare. Dans J. Prieto, A. K. Das, S. Ferretti, A. Pinto, & J. M. Corchado (Éds.). *Blockchain and Applications - International Congress (BLOCKCHAIN 2019), Avila, Spain*, pp. 36–44. doi: 10.1007/978-3-030-23813-1_5

Nakamoto, S. (2008). *Bitcoin : A peer-to-peer electronic cash system*. Repéré le 13 février 2020, à https://bitcoin.org/bitcoin.pdf

Nakov, S. (2018). Ethereum wallet encryption. Dans *Practical cryptography for developers* . Repéré à https://cryptobook.nakov.com/

Nandi, P., Koenig, D., Moser, S., Hull, R., Klicnik, V., Claussen, S., Kloppmann, M. & Vergo, J. (2010). *Data4BPM, part 1 : Introducing business entities and the business entity definition language (BEDL)*. Repéré le 17 février 2020, à https://pdfs.semanticscholar.org/6bd5/f30e6984821009a0c9f1691e26b059068d49.pdf

Nchinda, N., Cameron, A., Retzepi, K. & Lippman, A. (2019). Medrec : A network for personal information distribution. Dans *International Conference on Computing, Networking and Communications, ICNC 2019, Honolulu, HI*, pp. 637–641. doi: 10.1109/ICCNC.2019.8685631

Nigam, A. & Caswell, N. S. (2003). Business artifacts : An approach to operational specification. *IBM Systems Journal*, *42*(3), 428–445. doi: 10.1147/sj.423.0428

NIST (2001). Advanced Encryption Standard (AES). *NIST FIPS 197*. doi:

10.6028/NIST.FIPS.197

NIST (2017). *NIST policy on hash functions*. Repéré le 26 mai 2020, à https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions

OASIS Standard (2013). *Extensible access control markup language (XACML) version 3.0*. Repéré à http://docs.oasis-open.org/xacml/3.0/

Object Management Group (2011). *Business process model and notation specification version 2.0*. Repéré le 05 août 2020, à https://www.omg.org/spec/BPMN/2.0/

Object Management Group (2017). *Unified modeling language (UML), v2.5.1*. Repéré le 15 septembre 2020, à https://www.omg.org/spec/UML/2.5.1/PDF

O'Dwyer, K. J. & Malone, D. (2014). Bitcoin mining and its energy footprint. Dans *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014), Limerick, Ireland*, pp. 280–285. doi: 10.1049/cp.2014.0699

Ouafi, K. & Vaudenay, S. (2009). Pathchecker : An RFID application for tracing products in supply-chains. Dans L. Batina (Éd.). *The 5th Workshop on RFID Security (RFIDSec 2009), Leuven, Belgium*. Repéré à https://core.ac.uk/download/pdf/147952956.pdf

Parity Technologies (2019). *Aura - Authority round*. Repéré le 12 septembre 2019, à http://wiki.parity.io/Aura.html

Petri, C. A. (1962). *Communication with Automata*. (Thèse de doctorat). University of Bonn. Traduction anglaise par C.F. Greene, 1965. Repéré à http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri/doc/Petri-diss-engl.pdf

Petri, C. A. & Reisig, W. (2008). Petri net. *Scholarpedia*, *3*(4), 6477. doi: 10.4249/scholarpedia.6477

Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A. & Nguena-Timo, O. L. (2012). Runtime enforcement of timed properties. Dans S. Qadeer & S. Tasiran (Éds.). *Runtime*

Verification, Third International Conference, RV 2012, Istanbul, Turkey, Vol. 7687, pp. 229–244. doi: 10.1007/978-3-642-35632-2_23

Pnueli, A. (1977). The temporal logic of programs. Dans *18th Annual Symposium on Foundations of Computer Science, Providence, RI*, pp. 46–57. doi: 10.1109/SFCS.1977.32

Pnueli, A. (1979). The temporal semantics of concurrent programs. Dans G. Kahn (Éd.). *Semantics of Concurrent Computation, Proceedings of the International Symposium, Evian, France*, Vol. 70, pp. 1–20. doi: 10.1007/BFb0022460

Pnueli, A. (1986). Applications of temporal logic to the specification and verification of reactive systems : A survey of current trends. Dans J. W. de Bakker, W. P. de Roever, & G. Rozenberg (Éds.). *Current Trends in Concurrency, Overviews and Tutorials* (pp. 510–584). doi: 10.1007/BFb0027047

Pritzker, P. & May, W. (2015). SHA-3 standard : Permutation-based hash and extendable-output functions. *NIST FIPS 202*. doi: 10.6028/NIST.FIPS.202

Quinn, L. (2017). *Global Bitcoin mining now consumes as much electricity as Nigeria*. Repéré le 30 août 2019, à https://thehustle.co/bitcoin-mining-electricity

Radadiya, S. (2019). *A simple performance test and difference : Go v/s Java*. Repéré le 07 août 2020, à https://medium.com/@radadiyasunny970/a-simple-performance-test-and-difference-go-v-s-java-e6f29ad65293

Rifkin, J. (2014). *The zero marginal cost society : The Internet of Things, the collaborative commons, and the eclipse of capitalism*. New York, NY: St. Martin's Press.

Rivest, R. L. (1992). The MD5 message-digest algorithm. *RFC 1321*

Rivest, R. L., Shamir, A. & Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120–126. doi: 10.1145/359340.359342

Rosen, K. (2011). *Discrete mathematics and its applications* (7th ed.). New York, NY:

McGraw-Hill Education.

Roudjane, M., Rebaine, D., Khoury, R. & Hallé, S. (2018). Real-time data mining for event streams. Dans *22nd IEEE International Enterprise Distributed Object Computing Conference (EDOC 2018), Stockholm, Sweden*, pp. 123–134. doi: 10.1109/EDOC.2018.00025

Sandhu, R. S., Coyne, E. J., Feinstein, H. L. & Youman, C. E. (1994). Role-based access control : A multi-dimensional view. Dans *10th Annual Computer Security Applications Conference (ACSAC 1994), Orlando, FL*, pp. 54–62. doi: 10.1109/CSAC.1994.367293

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security*, *3*(1), 30–50. doi: 10.1145/353323.353382

Schneier, B. (1996). *Applied cryptography - Protocols, algorithms, and source code in C* (2nd ed.). Repéré à https://www.wiley.com/en-us/Applied+Cryptography%3A+Protocols%2C+Algorithms%2C+and+Source+Code+in+C%2C+2nd+Edition-p-9780471117094

Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. Dans *Proceedings of the First International Conference on Peer-to-Peer Computing, Linkoping, Sweden*, pp. 101–102. doi: 10.1109/P2P.2001.990434

Shanahan, C., Kernan, B., Ayalew, G., McDonnell, K., Butler, F. & Ward, S. (2009). A framework for beef traceability from farm to slaughter using global standards : An Irish perspective. *Computers and Electronics in Agriculture*, *66*(1), 62–69. doi: 10.1016/j.compag.2008.12.002

Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal*, *28*(4), 656–715. doi: 10.1002/j.1538-7305.1949.tb00928.x

Sharma, R. (2019). *Three people who were supposedly Bitcoin founder Satoshi Nakamoto*. Repéré le 30 août 2019, à https://www.investopedia.com/tech/three-people-who-were-supposedly-bitcoin-founder-satoshi-nakamoto/

Sobel, A. E. K. & Alves-Foss, J. (1999). A trace-based model of the Chinese wall security policy. Dans *Proceedings of the 22nd National Information Systems Security Conference,*

Arlington, VA. Repéré à https://csrc.nist.gov/csrc/media/publications/conference-paper/1999/10/21/proceedings-of-the-22nd-nissc-1999/documents/papers/p9.pdf

Subramanya, S. & Yi, B. (2006). Digital rights management. *IEEE Potentials*, *25*(2), 31–34. doi: 10.1109/MP.2006.1649008

Szabo, N. (1996). Smart contracts : Building blocks for digital markets. *Extropy : Journal of Transhumanist Thought*, *16*

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*, *2*(9). doi: 10.5210/fm.v2i9.548

Szilágyi, P. (2017). *Clique PoA protocol & Rinkeby PoA testnet*. Repéré le 12 septembre 2019, à https://github.com/ethereum/EIPs/issues/225

The European Parliament and the Council of the European Union (2016). *Regulation (EU) 2016/679 of the European parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free Movement of such data, and repealing directive 95/46/EC (General Data Protection Regulation)*. Repéré le 10 septembre 2020, à https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679

The Linux Foundation (2020). *Hyperledger fabric*. Repéré le 21 mai 2020, à https://www.hyperledger.org/projects/fabric

The London Pass (2020). *Transport for London*. Repéré le 29 octobre 2020, à https://www.londonpass.com/london-transport/

Toyoda, K., Mathiopoulos, P. T., Sasase, I. & Ohtsuki, T. (2017). A novel blockchain-based product ownership management system (POMS) for anti-counterfeits in the post supply chain. *IEEE Access*, *5*, 17465–17477. doi: 10.1109/ACCESS.2017.2720760

Vaculín, R., Hull, R., Heath, T., Cochran, C., Nigam, A. & Sukaviriya, P. (2011). Declarative business artifact centric modeling of decision and knowledge intensive business processes. Dans *Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2011), Helsinki, Finland*, pp. 151–160. doi: 10.1109/EDOC.2011.36

van der Aalst, W. M. P. (2016). *Process mining - Data science in action* (2nd ed.). doi: 10.1007/978-3-662-49851-4

Varvaressos, S., Lavoie, K., Gaboury, S. & Hallé, S. (2017). Automated bug finding in video games : A case study for runtime monitoring. *Computers in Entertainment*, *15*(1), 1 :1–1 :28. doi: 10.1145/2700529

Viriyasitavat, W., Da Xu, L., Bi, Z. & Sapsomboon, A. (2018). Blockchain-based business process management (BPM) framework for service composition in industry 4.0. *Journal of Intelligent Manufacturing*. doi: 10.1007/s10845-018-1422-y

Visa (2018). *Annual report*. Repéré le 20 juillet 2019, à https://s1.q4cdn.com/050606653/files/doc_financials/annual/2018/Visa-2018-Annual-Report-FINAL.pdf

Vranken, H. (2017). Sustainability of Bitcoin and blockchains. *Current Opinion in Environmental Sustainability*, *28*, 1–9. doi: 10.1016/j.cosust.2017.04.011

W3C (1999). *XML path language (XPath) Version 1.0*. Repéré le 19 juin 2020, à https://www.w3.org/TR/1999/REC-xpath-19991116/

W3C (2006). *Extensible markup language (XML) 1.1, 2nd Ed.* Repéré le 19 juin 2020, à https://www.w3.org/TR/xml11/

W3C (2007). *Web services description language (WSDL) Version 2.0 - Part 1 : Core language*. Repéré le 19 juin 2020, à https://www.w3.org/TR/wsdl20/

W3C (2012). *XML schema definition language (XSD) 1.1 - Part 1 : Structures*. Repéré le 19 juin 2020, à https://www.w3.org/TR/xmlschema11-1/

Wang, F., Zhou, C. & Nie, Y. (2013). Event processing in sensor streams. Dans C. C. Aggarwal (Éd.). *Managing and Mining Sensor Data* (pp. 77–102). doi: 10.1007/978-1-4614-6309-2_4

Wang, X. & Liu, N. (2014). The application of internet of things in agricultural means of production supply chain management. *Journal of Chemical and Pharmaceutical Research*,

6(7), 2304–2310

Wang, X. & Yu, H. (2005). How to break MD5 and other hash functions. Dans R. Cramer (Éd.). *Advances in Cryptology – EUROCRYPT 2005, Aarhus, Denmark*, pp. 19–35. doi: 10.1007/11426639_2

Wang, Y., Han, J. H. & Beynon-Davies, P. (2019). Understanding blockchain technology for future supply chains : A systematic literature review and research agenda. *Supply Chain Management : An International Journal*, *24*(1), 62–84. doi: 10.1108/SCM-03-2018-0148

Wilke, T. (1999). Classifying discrete temporal properties. Dans C. Meinel & S. Tison (Éds.). *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany*, pp. 32–46. doi: 10.1007/3-540-49116-3_3

Wood, G. (2019). *Ethereum : A secure decentralised generalised transaction ledger - Byzantium version 7e819ec*. Repéré le 13 février 2020, à https://ethereum.github.io/yellowpaper/paper.pdf

Wright, A., Andrews, H., Hutton, B. & Dennis, G. (2019). *JSON schema : A media type for describing JSON documents*. Repéré le 19 juin 2020, à https://json-schema.org/draft/2019-09/json-schema-core.html

Yardley, J. & Barboza, D. (2008). *Despite warnings, China's regulators failed to stop tainted milk*. Repéré le 27 juillet 2019, à https://www.nytimes.com/2008/09/27/world/asia/27milk.html

Yuan, E. & Tong, J. (2005). Attributed based access control (ABAC) for web services. Dans *2005 IEEE International Conference on Web Services (ICWS 2005), Orlando, FL*, pp. 561–569. doi: 10.1109/ICWS.2005.25

Zhang, Y., Kasahara, S., Shen, Y., Jiang, X. & Wan, J. (2019). Smart contract-based access control for the internet of things. *IEEE Internet of Things Journal*, *6*(2), 1594–1605. doi: 10.1109/JIOT.2018.2847705

Zhao, X., Su, J., Yang, H. & Qiu, Z. (2009). Enforcing constraints on life cycles of business artifacts. Dans W. Chin & S. Qin (Éds.). *Third IEEE International Symposium on*

Theoretical Aspects of Software Engineering (TASE 2009), Tianjin, China, pp. 111–118. doi: 10.1109/TASE.2009.46