HAL open science

Construction d'un multi-modèle d'application répartie pour la détection d'intrusion

David Lanoë

▶ **To cite this version:**

David Lanoë. Construction d'un multi-modèle d'application répartie pour la détection d'intrusion. Cryptographie et sécurité [cs.CR]. CentraleSupélec, 2020. Français. NNT : 2020CSUP0006 . tel-03564183

HAL Id: tel-03564183
https://theses.hal.science/tel-03564183v1

Submitted on 10 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

THÈSE DE DOCTORAT DE

CENTRALESUPELEC

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

David LANOË

Construction d'un multi-modèle d'application répartie pour la détection d'intrusion

Thèse présentée et soutenue à Rennes, le 15/12/2020
Unité de recherche : IRISA
Thèse N° : 2020CSUP0006

Rapporteurs avant soutenance :

Joaquin Garcia-Alfaro	Professeur, Telecom SudParis
Maria Potop-Butucaru	Professeur, Sorbonne Université - LIP6

Composition du Jury :

Président :	Vincent Nicomette	Professeur, Insa de Toulouse
Examinateurs :	Joaquin Garcia-Alfaro	Professeur - Telecom SudParis
	Maria Potop-Butucaru	Professeur, Sorbonne Université - LIP6
	Vincent Nicomette	Professeur, Insa de Toulouse
	Guillaume Doyen	Maître de conférences, Université de Technologie de Troyes
Dir. de thèse :	Michel Hurfin	Chargé de recherche, Inria Rennes-Bretagne Atlantique
Co-dir. de thèse :	Eric Totel	Professeur, IMT Atlantique Rennes

Invité(s) :

Frédéric Majorczyk	Docteur-Ingénieur, DGA-MI

Remerciements

Je tiens tout d'abord à remercier tous les membres du jury pour avoir accepté d'évaluer mes travaux. En particulier, je remercie Joaquin Garcia-Alfaro et Maria Potop-Butucaru qui ont accepté de rapporter cette thèse et Vincent Nicomette qui a accepté de présider le jury.

Je remercie ensuite ceux qui ont contribué à la fois scientifiquement et humainement à ce doctorat. Mes remerciements vont avant tout à mes deux directeurs de thèse Éric Totel et Michel Hurfin qui ont non seulement aidé à la réalisation de ce manuscrit au travers de leurs relectures mais qui ont également fait preuve de soutien, d'humour et de disponibilité tout au long de ma thèse. Mes remerciements vont aussi à Carlos Maziero qui a contribué à la réalisation de deux articles lors de son passage dans l'équipe CIDRE et à Frédéric Majorczyk pour les divers échanges ainsi que ses remarques sur le manuscrit.

Plus globalement, je remercie les membres actuels et passés de l'équipe CIDRE (qu'ils soient permanents ou non permanents) pour m'avoir accueilli et pour l'ambiance de travail très agréable.

Enfin, je remercie ma famille et mes amis pour leurs encouragements et leur soutien nécessaire à ce travail.

Résumé

L'étude réalisée durant cette thèse porte sur la sécurité des applications distribuées. Bien que les problèmes de sécurité soient désormais traités au cours du processus de développement des applications, le risque qu'une attaque puisse affecter les services fournis ou permettre l'accès à des données confidentielles ne peut être totalement exclu. Pour détecter les intrusions, nous considérons un mécanisme de détection d'anomalies qui repose sur un modèle du comportement normal de l'application surveillée. Au cours d'une phase de construction du modèle, l'application est exécutée plusieurs fois pour observer certains de ses comportements corrects. Chaque trace collectée permet d'identifier des événements significatifs et leurs relations de causalité, sans qu'une horloge globale soit nécessaire. Le modèle construit est double : il comporte un automate et une liste d'invariants probables qui caractérisent tous les deux des séquences d'événements autorisées. Les deux sous-modèles sont au départ redondants mais cette redondance diminue lorsque des techniques de généralisation sont appliquées à l'automate. Les solutions déjà proposées souffrent de problèmes de passage à l'échelle. En particulier, le temps nécessaire à la construction du modèle est important et, durant la phase de détection, la taille du modèle a une incidence sur la durée de l'analyse. Dans cette thèse, des solutions sont proposées pour résoudre ces problèmes de passage à l'échelle, tout en conservant une bonne précision pendant la phase de détection, en termes de taux de faux positifs et de faux négatifs. Pour évaluer les solutions proposées, une application distribuée réelle est utilisée et plusieurs attaques contre le service sont envisagées.

Table des matières

1 Introduction **1**

1.1 Enjeux de sécurité des applications distribuées . . . 1

1.2 La détection d'anomalies comme cadre d'études . . . 2

1.3 Objectifs de la thèse et contributions . . . 4

1.4 Organisation du document . . . 5

2 État de l'art **7**

2.1 Détection d'intrusion dans les applications distribuées . . . 7

2.1.1 Les approches de détection d'intrusion . . . 8

2.1.1.1 La méthode de détection . . . 8

2.1.1.2 Les sources de données . . . 9

2.1.2 La corrélation d'alertes . . . 10

2.1.3 Des approches comportementales parfois éloignées . . . 12

2.2 Quelques notions relatives aux applications distribuées . . . 12

2.3 Apprentissage d'un modèle de comportement . . . 18

2.3.1 Contexte . . . 19

2.3.2 Inférence d'automates . . . 19

2.3.3 Inférence d'invariants . . . 24

2.3.3.1 Logique temporelle . . . 24

2.3.3.2 Simplification des propriétés . . . 25

2.3.3.3 Inférence automatique de propriétés . . . 27

2.3.3.4 Inférence de propriétés pour les applications réparties . . 29

2.4 Conformité d'un modèle avec les comportements réels . . . 30

2.4.1 Complétude du modèle . . . 30

2.4.2 Précision du modèle . . . 32

2.5 Conclusion . . . 33

3 Une première approche pour la construction d'un modèle à partir de traces 35
3.1 Présentation Générale 35
3.2 Inférence d'un modèle intermédiaire à partir d'une seule trace 37
3.2.1 Construction du treillis des états globaux cohérents 37
3.2.2 Inférence d'automates 41
3.2.3 Inférence d'invariants 41
3.2.3.1 Evénements minimaux et événements maximaux 41
3.2.3.2 Nouvelles formulations des classes d'invariants 42
3.2.3.3 Coordonnées des états du treillis et relation de dépendance causale 46
3.2.3.4 Calcul des ensembles minimaux et maximaux 49
3.2.3.5 Inférence des invariants 54
3.3 Fusion de modèles intermédiaires et généralisation 56
3.3.1 Fusion d'automates et généralisation 56
3.3.2 Fusion d'invariants 59
3.4 Détection 65
3.5 Conclusion 71

4 Passage à l'échelle 73
4.1 Problèmes de passage à l'échelle 73
4.1.1 La croissance exponentielle du treillis 73
4.1.2 La généralisation d'automates de grande taille 74
4.2 Calcul d'un sous ensemble du treillis 75
4.3 Fusion et généralisation itérative 79
4.4 Conclusion 82

5 Évaluation 83
5.1 Cas d'étude n° 1 : Une application de commerce en ligne 83
5.2 Cas d'étude n° 2 : Un système de fichiers distribué 85
5.2.1 Présentation 85
5.2.2 Attaques contre le système de fichiers distribué 87
5.2.2.1 Attaque via le changement d'IP d'un OSD 87
5.2.2.2 Attaque sur la création de fichiers 87
5.2.2.3 Attaque sur la suppression de fichiers 89
5.2.2.4 Attaque sur les permissions d'accès 90
5.2.2.5 Attaque sur le changement de propriétaire 91

5.2.3 Collecte de données . . . 91
5.3 Évaluation des performances . . . 92
5.3.1 Passage à l'échelle lors de la phase de modélisation . . . 92
5.3.2 Impact d'un modèle dual sur le temps de détection . . . 94
5.4 Évaluation de la qualité du modèle . . . 96
5.4.1 Capacités de détection . . . 96
5.4.2 Complétude du modèle . . . 99
5.4.3 Acceptation de traces de tailles variables . . . 101
5.5 Conclusion . . . 102

6 Propositions de modèles complémentaires 105
6.1 Invariants . . . 105
6.1.1 Invariants d'occurrence . . . 105
6.1.1.1 Proposition . . . 106
6.1.1.2 Inférence d'invariants d'occurrence à partir d'une trace d'exécution . . . 106
6.1.1.3 Fusion des listes d'invariants d'occurrence . . . 108
6.1.1.4 Détection . . . 109
6.1.1.5 Évaluation . . . 109
6.1.2 Introduction de scopes pour les invariants . . . 112
6.1.2.1 Proposition . . . 112
6.1.2.2 Inference des portées . . . 113
6.1.2.3 Fusion et détection . . . 114
6.1.2.4 Évaluation . . . 115
6.2 N-grammes . . . 116
6.2.1 Proposition . . . 116
6.2.2 Inférence de N-grammes . . . 117
6.2.3 Fusion de N-grammes et construction de l'automate . . . 118
6.2.4 Différences avec l'automate généralisé et détection . . . 120
6.2.5 Évaluation . . . 121
6.3 Conclusion . . . 123

7 Conclusion et travaux futurs 125
7.1 Contributions . . . 125
7.2 Perspectives . . . 127

A Liste des publications 129

Bibliographie 131

Chapitre 1

Introduction

1.1 Enjeux de sécurité des applications distribuées

En tant qu'utilisateur, nous sommes de plus en plus concernés par les applications de stockage et partage de documents, de commerce électronique, de banque en ligne, de jeux en ligne, etc. Via nos ordinateurs, tablettes et smartphones, nous interagissons régulièrement avec des serveurs, des bases de données et des logiciels applicatifs mis à notre disposition. L'essor de l'internet des objets renforce encore notre immersion dans un monde numérique dématérialisé. Les traitements que nous déclenchons ou avec lesquels nous interagissons impliquent une coopération entre des calculateurs distants. Dans ces systèmes distribués parfois large échelle et parfois extrêmement complexes, les problématiques de sécurité ne peuvent pas être négligées. Elles sont désormais prises en compte dès les phases de conception et développement. Mais, malgré tout les efforts de prévention, il est difficile d'assurer que l'exécution d'une application résistera à toute sorte d'actions malveillantes. Des vulnérabilités (identifiées ou pas) au niveau du logiciel ou du matériel peuvent être exploitées par un attaquant. Des applications sensibles telles que le commerce électronique ou les systèmes de fichiers qui s'exécutent désormais sur des systèmes distribués à grande échelle sont donc potentiellement vulnérables. Elles constituent des cibles privilégiées pour les attaquants susceptibles d'utiliser le service à mauvais escient, de voler des informations, de compromettre la disponibilité du service ou l'intégrité des données. Une seconde ligne de défense est nécessaire pour contrer les attaquants qui réussissent (ou sont sur le point de réussir) à corrompre un système d'information. L'objectif est d'identifier qu'une intrusion est en cours (ou vient de réussir) afin de lever une alerte et de permettre aux administrateurs de prendre des contre-mesures pour protéger le système ou limiter les dommages causés.

1.2 La détection d'anomalies comme cadre d'études

L'approche habituelle pour détecter les comportements incorrects dans les systèmes distribués consiste à surveiller chaque processus (ou chaque élément du réseau de communication) avec un système de détection d'intrusion, en anglais : IDS (Intrusion Detection System). Un IDS observe des activités locales et génère des informations sur ce qu'il observe (occurrences d'actions particulières) ou sur ce qu'il interprète comme un comportement localement suspect (alertes bas niveau). Tout ces événements générés localement par les différents IDS sont estampillés grâce à une horloge globale et envoyés à un moteur de corrélation central. Ce moteur ordonne tous les événements en utilisant leurs horodatages et analyse la séquence ainsi obtenue afin de détecter des motifs prédéfinis correspondant à des scénarios d'attaque [LHT18]. Cette approche, qui est connue sous le nom d'approche par scénarios (ou approche par signatures), ne peut détecter que des attaques connues, pour lesquelles une «signature» d'événements a pu être spécifiée. De plus, une horloge globale est nécessaire pour construire l'ordre total sur tous les événements, ce qui peut être impossible notamment dans le cas de grandes applications distribuées.

L'étude réalisée durant cette thèse s'est focalisée sur une approche alternative : la détection d'anomalies. Les systèmes de détection d'intrusion destinés à détecter des anomalies recherchent des écarts par rapport à un modèle de référence du comportement normal ; tout écart est interprété (à tort ou à raison) comme une conséquence observable d'une attaque. Aucune connaissance préalable des attaques possibles n'est requise. Par contre, il est nécessaire d'avoir un modèle permettant d'identifier quels sont les comportements normaux de l'application.

A moins de disposer d'une spécification du comportement normal de l'application (ce qui est malheureusement assez rare), il faut inférer cette connaissance durant une phase dite de construction du modèle. Une solution consiste à exécuter l'application à plusieurs reprises dans un environnement sécurisé (sans attaquant) afin de collecter de l'information sur ses différents comportements normaux. La connaissance acquise durant cet apprentissage n'est que partielle puisqu'il est souvent impossible de découvrir l'ensemble des comportements autorisés durant un nombre fini d'exécutions. Le modèle construit identifie des contraintes sur l'enchaînement des événements durant une exécution normale. Son rôle est de permettre d'identifier si une séquence d'événements est valide ou ne l'est pas.

Différentes représentations (sous la forme d'un automate, sous la forme d'une liste d'invariants,...) peuvent être utilisées. Le modèle peut se fonder sur une seule représentation ou au contraire être composé de plusieurs sous-modèles distincts. Il peut refléter

fidèlement ce qui a été appris durant l'apprentissage et n'accepter que les comportements appris. Mais, dans ce cas, tout comportement normal non appris est susceptible d'être ensuite rejeté par l'outil de détection car il ne sera pas nécessairement reconnu comme valide par le modèle. Pour éviter cela, des techniques dite de généralisation transforment le modèle (souvent en le réduisant) pour qu'il soit moins restrictif et accepte également des comportements non appris. Une technique de généralisation est donc indispensable pour réduire le nombre de faux positifs (alertes générées sans raison). Mais, parmi les comportements non appris qu'une technique de généralisation permet d'autoriser, certains peuvent être des comportements symptomatiques d'une attaque : une technique de généralisation qui vise à réduire le nombre de faux positif peut donc aussi potentiellement engendrer des faux négatif (une attaque réelle ne sera pas détectée). En matière de modélisation des comportements normaux, il existe donc un vaste panel de solutions possibles et seules quelques unes ont été proposées et évaluées à ce jour.

Une fois construit, le modèle peut être utilisé par le mécanisme de détection d'intrusion. Ce mécanisme est en charge de surveiller l'application qui s'exécute désormais dans un environnement non sécurisé. Durant cette phase dite de détection, les informations collectées sont exploitées pour déterminer si l'exécution est conforme ou pas au modèle de l'application. En cas de divergence, une alerte est levée.

Dans l'approche dont les grandes lignes viennent d'être décrites, il est nécessaire de pouvoir observer le comportement d'une application durant une exécution. Cela se fait aussi bien durant la phase de construction du modèle (pour apprendre plusieurs comportements) que durant la phase de détection (pour surveiller un comportement). Une particularité de notre approche est que lors de ces deux phases qui visent respectivement à créer et exploiter un modèle, il n'est pas nécessaire d'observer l'ordre total dans lequel les événements se produisent. Comme nous l'avons indiqué auparavant (au moment de la description de l'approche par scénarios), une observation précise de l'ordre total entre événements requiert de disposer d'une horloge globale. L'approche par détection d'anomalies est capable de s'appuyer sur une connaissance plus approximative de cet ordre (via une relation d'ordre partiel comme la relation "happen before" de Lamport [Lam78]). Le fait que cette relation soit moins problématique à observer est le principal avantage. Il existe également un deuxième avantage : lors de la phase d'apprentissage, la relation d'ordre partiel permet de capturer plusieurs ordres totaux compatibles qui correspondent chacun à un comportement possible de l'application. De fait, au lieu d'apprendre un seul comportement par exécution, c'est un ensemble de comportements ayant comme point commun de partager la même relation d'ordre partiel qui sont appris en une seule exécution.

1.3 Objectifs de la thèse et contributions

La première étape du travail de thèse a consisté à spécifier et développer un premier outil de détection d'anomalies reposant sur une combinaison raisonnable de solutions déjà proposées dans la littérature (mais pas toujours évaluées). Ce travail nous a notamment conduit à faire les trois choix pérennes suivants. Premièrement, chaque exécution est considérée comme un ensemble partiellement ordonné d'événements. Les deux principaux avantages attendus viennent d'être soulignés. Le fait de pouvoir se passer d'une horloge globale est un atout indéniable, notamment quand on ne dispose pas d'une telle horloge. Par contre, le fait de considérer tous les ordres totaux compatibles avec un ordre partiel permet certes d'avoir un modèle plus général mais cela a également un coût. Deuxièmement, le modèle est supposé pouvoir être composé de plusieurs sous-modèles distincts. Deux sous modèles (l'un correspondant à un automate et l'autre à une liste de pseudo invariants) ont été considérés durant l'étude mais d'autres sous-modèles ont aussi été envisagés. La gestion simultanée de plusieurs modèles a un coût : seule une complémentarité des sous-modèles peut justifier un tel choix. Troisièmement, la structuration modulaire du logiciel qui a été adoptée permet de bien distinguer la phase de généralisation du modèle des phases initiales de création des modèles. L'objectif est de pouvoir venir par la suite intégrer différentes techniques de généralisation.

Le principal défi de cette approche est de construire un modèle efficace tout en prenant en compte l'évolutivité (lors de la construction du modèle et des phases de détection). Notre objectif ultime étant de détecter les anomalies, l'efficacité peut être mesurée en termes de taux de faux positifs et de faux négatifs. Le temps nécessaire à la création et à la mise à jour du modèle est important, en particulier pour les traces d'exécution plus longues. Enfin, les modèles plus petits permettent une détection plus rapide des attaques, ce qui est souvent une exigence majeure.

La contribution de cette thèse est quadruple : 1) Alors que le treillis des coupures cohérentes croît de façon exponentielle dans le contexte de systèmes distribués de grande taille [Gar12a], nous proposons des solutions qui réduisent le temps de calcul et génèrent un modèle initial plus compact. Nous montrons qu'une sélection judicieuse de quelques séquences (parmi l'ensemble des ordres totaux compatibles) nous permet d'obtenir plus rapidement un modèle non moins efficace ; 2) Les solutions existantes construisent généralement un modèle énorme et complexe basé sur un automate, puis le réduisent au cours d'une phase de généralisation finale. Nous proposons une approche itérative où le modèle en construction est à plusieurs reprises fusionné et généralisé, en prenant en compte, à chaque étape, un modèle intermédiaire spécifique à l'une des exécutions apprises ; 3) Pour analyser leurs impacts sur les performances mais également sur la qualité

de la détection, les solutions proposées ont été mises en œuvre et évaluées à l'aide d'une application existante gérant un système de fichiers distribué et en concevant de véritables attaques contre celui-ci ; 4) enfin, nous montrons que les deux sous-modèles (décrits par une liste de pseudo-invariants et un automate) sont complémentaires, car ils ne détectent pas exactement les mêmes anomalies.

Au cours de cette thèse, différents développements ont été réalisés. Ils peuvent se diviser en trois catégories :

— Les développements mettant en œuvre notre approche et nos propositions : On retrouve notamment dans cette catégorie, les briques logicielles permettant la construction du modèle de comportement à l'aide de traces d'exécutions ainsi que le module de détection d'attaques. Les algorithmes liés à ces outils sont détaillés dans les chapitres 3, 4 et 6.

— Les développements permettant l'évaluation via des applications de l'approche et des différentes propositions : La validation des contributions a nécessité l'instrumentation du code d'applications distribuées pour la collecte de traces, l'élaboration de scripts pour effectuer des actions légitimes et collecter des traces correctes et la mise en œuvre d'attaques pour la collection de traces incorrectes. Ces développements sont peu détaillés dans le cadre de ce manuscrit. Ils sont mentionnés au chapitre 5.

— Enfin, les développements pour faciliter l'analyse des différentes données obtenues : Les outils développés dans ce contexte avaient pour objectif de traiter les données d'expérimentations pour obtenir des statistiques et d'un outil de visualisation d'automates. Ces outils ne sont pas détaillés dans ce manuscrit. L'outil de visualisation est cité dans le chapitre 6.

1.4 Organisation du document

En plus de ce court chapitre introductif et d'un chapitre de conclusion, le manuscrit de thèse comporte cinq autres chapitres. Le chapitre 2 est dédié à un état de l'art sur les thèmes couverts. Le chapitre 3 décrit les différents composants d'une solution de détection d'intrusion (inspirée de l'état de l'art) que j'ai entièrement développé. Cette solution sert de point de référence dans les évaluations qui sont réalisées. Le chapitre 4 présente des solutions apportées pour résoudre le problème du passage à l'échelle. Le chapitre 5 est dédié à l'évaluation des solutions proposées. Le chapitre 6 aborde d'autres formes de modélisation.

Chapitre 2

État de l'art

Dans ce chapitre entièrement consacré à l'état de l'art, une première partie (section 2.1) présente des notions générales sur le thème de la détection d'intrusion. La thèse se focalise sur la détection d'anomalie. Néanmoins, afin de positionner les travaux de cette thèse, les différences avec la détection par signature sont précisées et les mécanismes de corrélation d'alertes sont brièvement abordés. Puisque les applications distribuées sont la cible de cette étude, une seconde partie (section 2.2) est consacrée à la représentation du comportement d'une application distribuée sous la forme d'un ensemble partiellement ordonné d'événements. Des notions et des résultats classiques dans le domaine des calculs distribués sont rappelés. Une troisième partie (section 2.3) aborde plus spécifiquement le thème de la détection d'anomalie et présente des approches et des outils utilisés pour inférer un modèle de comportement à partir de traces d'exécution. Enfin, une quatrième et dernière partie (section 2.4) identifie des problématiques liées à la conformité des modèles de comportements inférés aux comportements des applications.

2.1 Détection d'intrusion dans les applications distribuées

Cette première partie est donc consacrée à la détection d'intrusion en général. Les différences entre les principales approches sont explicitées et notre axe de recherche est peu à peu précisé. La prise en compte d'attaques complexes via des moteurs de corrélation est brièvement décrite. Puisque l'objectif de cette partie est aussi de faire un tour d'horizon rapide des domaines de recherche connexes mais sans lien fort avec nos travaux, nous terminons en mentionnant les approches comportementales fondées sur l'existence d'une spécification.

2.1.1 Les approches de détection d'intrusion

La détection d'intrusion est une problématique de sécurité qui a commencé à faire l'objet de nombreux travaux dans les années 1980 [Den87] avec l'apparition à cette époque de plusieurs prototypes [Sma88, LJ88]. En 1999, Debar [DDW99] propose une taxonomie des systèmes de détection d'intrusion (IDS, Intrusion Detection System) qui est reprise ici pour structurer la rédaction de cette partie.

La sécurisation d'un système d'information commence par la définition d'une politique de sécurité des systèmes d'information (PSSI) [ANS04]. Cette dernière constitue le principal document de référence en matière de SSI d'un organisme. Elle a pour objectif d'identifier les différents éléments à protéger (matériel, immatériel, informations relatives aux personnes) à l'aide d'une analyse de risque et de définir les moyens techniques ou organisationnels pour mettre en œuvre cette politique de sécurité.

Les systèmes de détection d'intrusion (IDS) visent à identifier des attaques qui sont des violations de la politique de sécurité du système d'information. Un IDS est un mécanisme destiné à identifier des activités suspectes ou anormales. Pour cela, il surveille dynamiquement les actions réalisées sur l'environnement ciblé. Son rôle ne se cantonne pas à observer. Il analyse les informations collectées afin de décider si ces actions relèvent d'une attaque ou si elles correspondent à une utilisation légitime de l'environnement.

Un IDS peut être classifié à l'aide de certaines de ses caractéristiques fonctionnelles :

- Le comportement de l'IDS lors de la détection : l'IDS peut soit seulement lever des alertes soit également réagir à celles-ci (IPS, intrusion prevention system) ;
- La méthode de détection utilisée par l'IDS ;
- Les sources de données que l'IDS collecte.

Le premier point n'est pas essentiel dans le cadre de cette étude puisque celle-ci se focalise uniquement sur les analyses amont qui mènent à la décision de lever ou pas une alerte. La réaction (manuelle ou partiellement automatisée) suite au lever d'une alerte ne fait pas partie du champs d'investigation couvert durant cette thèse. Nous nous intéresserons donc plus particulièrement aux méthodes de détection et aux sources de données.

2.1.1.1 La méthode de détection

La méthode de détection correspond à l'approche choisie par l'IDS pour traiter l'information collectée à propos des actions observées. L'approche choisie peut être soit une approche par signature soit une approche comportementale. Lorsqu'un IDS utilise la connaissance des attaques auxquelles le système est exposé, il utilise une approche par signature. Quand il cherche à identifier les déviations par rapport à un modèle de

comportement normal du système, il utilise une approche par comportement. Ces deux approches sont complémentaires.

L'approche par signature utilise des connaissances préalablement acquises concernant des attaques spécifiques et des vulnérabilités du système. Dans cette approche, un IDS détient des informations à propos des vulnérabilités du système et essaye d'identifier les tentatives d'exploitation de celles-ci. Quand une de ces tentatives est détectée, une alerte est levée. Autrement dit, toute action non reconnue comme une attaque est considérée comme saine. Ce type d'approche permet d'avoir une vision détaillée des attaques qui ont eu lieu. Cependant, la détection par ce type d'approche ne considère pas la détection d'attaques inconnues. D'autre part, il est nécessaire de collecter des informations sur le fonctionnement du système et de décrire les vulnérabilités de celui-ci.

L'approche comportementale considère qu'une intrusion peut être détectée en observant une déviation du comportement normal. Le modèle de comportement normal est extrait des informations de référence collectées par différents moyens. Ce modèle est ensuite comparé aux activités en cours. Si une déviation est observée, une alerte est générée. Autrement dit, tout ce qui ne correspond pas au modèle préalablement construit est considéré comme une attaque. Un IDS utilisant ce type d'approche va permettre de détecter des attaques nouvelles et non connues sans avoir besoin de mettre régulièrement à jour le modèle. Cependant, dans le cas où le comportement normal du système évolue (à la suite par exemple d'une évolution d'un de ses composants), le modèle doit impérativement être mis à jour puisqu'il fait explicitement référence aux actions normales des différents composants (un appel système, un appel de fonction, l'exécution d'une instruction particulière, ...). En l'absence d'information précise sur le système ou l'application, des comportements normaux peuvent être identifiés en observant le déroulement du calcul durant plusieurs exécutions successives. La construction d'un modèle de comportement normal nécessite qu'aucune attaque n'ait lieu pendant cette phase d'apprentissage.

Dans le contexte des applications distribuées, nous avons fait le choix d'explorer une approche de type comportementale. Ce type d'approche ne nécessitant pas de connaître le fonctionnement du système et des différentes vulnérabilités l'impactant, il est possible de l'appliquer à des applications distribuées quelconques dès lors que la construction d'un modèle caractérisant les comportements normaux est envisageable.

2.1.1.2 Les sources de données

Les sources de données correspondent aux données que l'IDS utilise lors de la phase de détection pour identifier si une attaque a eu lieu. L'accès aux différentes sources de données va dépendre de l'endroit où est positionné l'IDS. Les données peuvent se situer

soit au niveau des hôtes (HIDS, Host Based Intrusion Detection System) soit au niveau du réseau (NIDS, Network Based Intrusion Detection System). Suivant la source de données utilisée, l'IDS pourra être plus efficace pour détecter certains types d'attaques plutôt que d'autres.

Les HIDS surveillent les informations liées aux activités des utilisateurs sur une machine. Pour cela, ils peuvent collecter et analyser des logs applicatifs, des logs systèmes, des fichiers systèmes ou les trafics entrant et sortant [SBDB00, GTM09]. Leur positionnement permet de collecter des informations précises sur des attaques visant une partie du système. Ils sont donc particulièrement efficaces pour détecter les attaques ciblant une machine précise. Cependant, cette famille d'IDS est vulnérable aux altérations des données. En cas d'attaque, l'attaquant peut modifier les données nécessaires au bon fonctionnement du HIDS ou prendre le contrôle de celui-ci. Les HIDS sont donc soumis à des contraintes de temps-réel. D'autre part, ils utilisent les ressources de calcul de l'hôte pour analyser les données.

Les NIDS comme Bro [Pax99] sont installés à des points particuliers du réseau. Ils surveillent le trafic entre plusieurs machines. Ils analysent principalement les informations provenant du protocole SNMP et les paquets réseaux. Cette famille d'IDS est adaptée pour détecter des attaques orientées réseau (DNS spoofing, TCP hijacking, port scanning, ping of death, ...). Ces types d'attaques sont difficilement détectées par les HIDS. Les NIDS peuvent être installés à des endroits qui n'affectent pas les performances du réseau et sont invisibles pour les attaquants (miroir de port ou TAP). Cependant, les NIDS peuvent se révéler peu efficaces pour détecter certaines attaques en cas de chiffrement des communications.

Dans le cas d'applications distribuées, nous avons choisi d'étudier les logs applicatifs. Ces derniers sont collectés localement sur chaque processus et sont liés au fonctionnement d'une même application. Dans l'approche comportemental, c'est la définition du modèle global qui permet d'établir des relations entre des données provenant de plusieurs sources distantes. Dans les approches par signature, la prise en compte du lien entre des informations provenant de différente sources est également un problème comme nous allons le voir maintenant au travers d'une brève description des mécanismes de corrélation d'alertes.

2.1.2 La corrélation d'alertes

Dans le cas où les données proviennent de différentes sources, la mise en relation de données se fait traditionnellement lors d'une phase de corrélation d'alertes.

Les différentes sondes réparties dans le système produisent un flux d'information (alertes bas niveaux et événements) qui ne peut être transmis directement à un administrateur. En effet celui-ci serait rapidement noyé sous une multitude d'informations pas toujours significatives, c'est à dire sans rapport avec une attaque ou difficilement interprétable. L'utilisation d'un processus de corrélation d'alertes [VVKK04] vise à réduire le volume d'alertes produit en centralisant et analysant de façon automatique l'information produite par les différentes sondes.

Le premier objectif est de diminuer le flux d'alertes généré (corrélation bas niveau). Pour cela, les alertes vont être pré-traitées, homogénéisées, vérifiées, agrégées et enrichies. En sortie de ce premier niveau de corrélation, la taille du flux est réduite car des éléments du flux ont été supprimés lors de vérifications tandis que d'autres ont été fusionnés : d'une part, plusieurs alertes bas niveaux peuvent être générées suite à l'observation d'une même action par plusieurs sondes et, d'autre part, plusieurs alertes successives peuvent faire référence à une même action (un scan de port par exemple). Même si l'information est plus riche et explicite, chaque élément du nouveau flux correspond toujours à une action élémentaire observée.

Le deuxième objectif de la corrélation d'alertes est de reconnaître les enchaînements d'alertes caractéristiques d'une intrusion (corrélation haut niveau). Les attaques réalisées sont de plus en plus complexes. Dans un système distribué, elles nécessitent parfois que l'attaquant exploite successivement plusieurs vulnérabilités sur plusieurs sites distants. De plus, ces étapes élémentaires dans l'attaque doivent souvent être réalisées dans un ordre particulier. La corrélation haut niveau se fait via la définition de scénarios d'attaques [TVM04, Val06]. Ces scénarios identifient un ensemble d'actions et des contraintes d'ordre entre certaines de ces actions. Cette description (par exemple, sous la forme d'un arbre d'attaque) permet ensuite de dériver des règles de corrélation qui sont en quelque sorte des descriptions des motifs (suites d'actions) qui devront être identifiés dans le flux. Alors que l'arbre d'attaque décrit l'enchaînement des actions d'un attaquant, la règle de corrélation décrit un motif correspondant à l'enchaînement des réactions des observateurs charger de défendre le système (sondes et autre). Une reconnaissance de scénarios d'attaques par un mécanisme de corrélation de haut niveau implique donc de connaître au préalable le fonctionnement du système (topologie, nature des sondes, vulnérabilités, ...) ainsi que les attaques contre celui-ci.

Dans le cas d'une approche par signature, la règle de corrélation fait référence à un scénario anormal. Nous verrons dans cette thèse que des modèles utilisés pour décrire des comportements normaux peuvent aussi parfois être exploités pour identifier des scénarios anormaux. De fait les moteurs de corrélation peuvent également être utilisés dans le cadre d'une approche comportementale.

2.1.3 Des approches comportementales parfois éloignées

Dans ce travail de thèse, nous mettons en place une approche comportementale. Cependant, nous tenons a insister ici sur le fait que cette classe très générale regroupe des travaux divers et variés où les problématiques ne sont pas toujours très proches. Ainsi des hypothèses faites dans cette étude nous démarque des travaux de Ko [KRL97] qui propose en 1997 une approche comportementale où le fonctionnement normal du système est décrit par des spécifications (règles de sécurité) et où un comportement est représenté par une séquence d'événements. Dans ce travail, les traces sont transférées et ordonnées au niveau du moniteur. Des alertes sont levées en cas de violation des spécifications, ces violations étant souvent causées par la mauvaise synchronisation des processus.

Indépendamment de l'intérêt des travaux présentés par Ko [KRL97], deux hypothèses différencient fortement leur travaux de ceux décrits dans ce manuscrit. D'une part, leur approche repose sur l'hypothèse d'un ordonnancement total des évènements à l'aide d'un mécanisme d'horloge globale. A l'inverse, dans nos travaux, le comportement observé durant une exécution est représenté par un ordre partiel entre les événements. D'autre part, le modèle que nous proposons, contrairement à celui exposé dans l'article [KRL97], n'est pas décrit grâce à un langage de spécification mais est inféré lors d'une phase d'apprentissage.

2.2 Quelques notions relatives aux applications distribuées

Une application distribuée implique plusieurs processus qui, en s'exécutant sur des machines distantes, coopèrent à la réalisation d'un objectif applicatif commun.

Considérons une application distribuée non déterministe notée A. Le non déterminisme est dû au fait que le code d'une application autorise en général plusieurs comportements possibles. Durant chaque exécution, un des comportements est suivi mais ce n'est pas nécessairement le même à chaque fois. Les variations s'expliquent notamment par le fait que les requêtes externes reçues ou les données fournies en entrée peuvent changer d'une exécution à une autre. La possibilité de définir aléatoirement le choix de la prochaine action à exécuter peut aussi être une source de non-déterminisme. Enfin, l'absence de synchronisme entre sites distants fait que les actions concurrentes (non dépendantes l'une de l'autre) se produisent dans un ordre non maîtrisé : en conséquence, l'ordre de consommation des messages sur un site peut varier.

Nous supposons que n est le nombre total de processus intervenant durant l'exécution de l'application A. Chaque processus est noté p_i avec $1 \leq i \leq n$ et l'ensemble des processus est noté $\Pi A = \{p_1, p_2, ..., p_n\}$. Lors d'une exécution de l'application, les

n processus vont chacun exécuter un code local et communiquer entre eux en se transmettant des messages. Pour connaître le comportement de l'application A durant son exécution, il est nécessaire de collecter de l'information sur les événements qui se produisent. Une identification des événements pertinents et susceptibles d'être observés doit donc être réalisée au préalable. Cette identification peut s'appuyer sur la notion de type d'événement sur laquelle nous reviendrons par la suite. Pour chacun des événements présentant un intérêt, de l'information décrivant chaque occurrence est collectée au cours de l'exécution et stockée dans le log local. Pour cela, il peut être nécessaire d'utiliser des dispositifs d'observation extérieurs ou d'instrumenter directement le code de l'application [TJC+98]. La collecte d'information et les problèmes associés tels que par exemple les effets de sonde [Gai86] ne sont pas détaillés dans ce manuscrit.

Durant une exécution observée, chaque processus génère son propre log et les n logs ainsi créés constituent une trace. Afin de distinguer les traces obtenues au cours d'exécutions différentes, les notations α, β, ... sont utilisées pour nommer les exécutions successives. La trace obtenue au cours d'une exécution α est notée E^α. Elle contient n fichiers de logs (un par processus). Le log produit par le processus p_i est noté E_i^α. Il contient un nombre fini d'événements : sa taille est notée l_i^α. Cet historique où les événements d'un même processus apparaissent chronologiquement est un ensemble d'événements totalement ordonné. De fait, chaque occurrence d'événement observé sur le site p_i peut être notée e_i^x où x correspond à l'indice de l'événement dans le fichier de log qui peut lui-même être représenté de la façon suivante : $E_i^\alpha = \{e_i^1, ..., e_i^{l_i^\alpha}\}$. Lorsque ni le site où se produit l'événement (p_i), ni le numéro dans la séquence local (x) ne sont utiles, nous utilisons les notations simples e, f ou g pour désigner des événements. Chaque événement qui se produit sur un site p_i peut être un événement interne (par exemple, un appel système) ou un événement de communication. Un événement de communication fait référence d'une part à un canal unidirectionnel (noté c) entre p_i et un autre processus p_j et d'autre part à un message m. Un événement de communication peut soit correspondre à l'envoi d'un message m sur un canal c, noté c !m ou à la réception d'un message m sur un canal c, noté c ? m.

La trace E^α est un ensemble d'événements regroupant tous les événements observés lors de l'exécution α : $E^\alpha = \cup_{i \in \{1,...,n\}} E_i^\alpha$. Le nombre d'événements de la trace est noté l^α. Par définition, $l^\alpha = \mid E^\alpha \mid = l_1^\alpha + l_2^\alpha + \cdots + l_n^\alpha$. Alors que chaque ensemble E_i^α est totalement ordonné, l'ensemble E^α n'est que partiellement ordonné par la relation « happened-before » [Lam78] qui est notée $\prec^\alpha$. Si la trace E^α contient tous les événements de communication, cette relation d'ordre partielle peut être définie comme suit.

Définition 1 *$\forall e \in E^\alpha$, $\forall f \in E^\alpha$, $e \prec^\alpha f$ si et seulement si :*

— *e apparaît avant f dans le même fichier de log* E_i^α*, ou*

— *e correspond à l'envoi d'un message et f correspond à la réception du même message, ou*

— $\exists\ g \in E^\alpha$ *tel que* $e \prec^\alpha g$ *et* $g \prec^\alpha f$

L'ensemble des événements E^α muni de la relation $\prec^\alpha$ est parfois appelé poset (de l'anglais "partially ordered set"). La relation $\prec^\alpha$ capture un sur-ensemble de la causalité entre les différents événements : $e \prec^\alpha f$ signifie que e s'est produit avant f durant l'exécution α.

Les couples d'événements qui ne sont pas ordonnés par la relation $\prec^\alpha$ sont dits concurrents (relation binaire $\|^\alpha$) : aucun ne peut affecter causalement l'autre.

Définition 2 $\forall e \in E^\alpha$*,* $\forall f \in E^\alpha$*,* $e \|^\alpha f$ *si et seulement si les deux relations* $e \prec^\alpha f$ *et* $f \prec^\alpha e$ *ne sont pas établies.*

Lors de l'observation d'une exécution, la notion de type d'événement permet d'avoir un premier niveau d'abstraction. A chaque événement exécuté e est associé un type d'événement que nous noterons $\overline{e}$. Souvent la définition d'un type d'événement permet de caractériser une classe d'événements correspondant à des actions exécutées ayant des caractéristiques proches. La notion de type d'événement se confond alors avec celles de type d'action et de type de message. Mais rien n'empêche d'adopter des définitions de type d'événement qui soient indépendantes de la nature des actions associées. Par exemple, il est possible de définir des événements de type pair et des évènements de type impair selon que l'identité de l'événement observé peut s'exprimer sous la forme e_i^{2k} ou e_i^{2k+1}. Nous utiliserons les notations a, b, c pour faire référence à des types d'événements quelconques. En général, dans les exemples décrits dans ce manuscrit, les types d'événements ne seront pas définis formellement. Pour éviter de multiplier inutilement les notations, la notation $c!m$ (respectivement $c?m$) sera utilisée également pour désigner le type d'événement correspondant à un envoi (respectivement une réception) d'un message de type m sur le canal c. Si a est un type d'événement alors l'ensemble des événements de type a qui se produisent au cours de l'exécution α est noté $E^\alpha(a)$. Autrement dit, $E^\alpha(a) = \{e \in E^\alpha \mid \overline{e} = a\}$. Enfin, l'ensemble $\overline{E^\alpha}$ correspond à l'ensemble de tous les types d'événements associés à des événements qui se sont produits durant l'exécution α.

Illustrons ces notations et définitions avec l'exemple d'une application A qui implique deux processus p_1 et p_2. L'application est observée durant deux exécutions distinctes α et β. Dans le tableau 2.2, les deux traces E^α et E^β sont décrites. Au lieu de faire figurer l'identité des événements qui se sont produits, ce sont leur types qui sont précisés dans

le tableau. La trace E^α générée au cours de l'exécution α est composée de deux fichiers de logs notés E_1^α et E_2^α. Dans cette exécution en particulier, les fichiers de logs de p_1 et de p_2 contiennent respectivement quatre et trois événements : $l_1^\alpha = 3$ et $l_2^\alpha = 4$. A noter que deux événements, à savoir e_1^1 et e_2^4 sont du même type b bien qu'ils se produisent sur des processus différents. Nous ignorons dans un premier temps l'exécution β qui illustre le non-déterminisme de l'application A.

Execution α		Execution β	
E_1^α	E_2^α	E_1^β	E_2^β
b	a	b	c
c1 !m	c2 !m	c1 !m	c2 !m
c2 ?m	c1 ?m	c2 ?m	c1 ?m
	b	d	

TABLEAU 2.1 – Trace correspondant aux exécutions α et β

Lors de l'exécution α, le processus p_1 exécute une action locale de type b, envoie le message m_1 au processus p_2 via le canal $c1$ et reçoit le message m_2 sur le canal $c2$. Parallèlement, le processus p_2 produit une action locale de type a, envoie le message m_2 au processus p_1 via le canal $c2$, reçoit le message m_1 venant du processus p_1 via le canal $c1$ et enfin produit une action locale de type b. Durant l'exécution α, les trois événements e_2^1, e_1^2 et e_1^3 tels que $\overline{e_2^1} = a$, $\overline{e_1^2} = c1!m$ et $\overline{e_1^3} = c2?m$ vérifient (entre autres) les relations suivantes $e_2^1 \prec^\alpha e_1^3$ et $e_2^1 \parallel^\alpha e_1^2$.

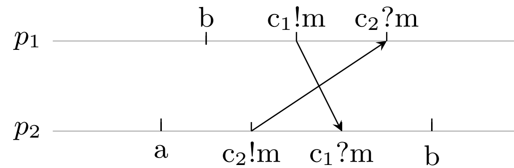

FIGURE 2.1 – Ordre total possible de l'exécution α

Si on fait l'hypothèse que l'exécution d'un événement est une action atomique instantanée, les sept événements se sont produits durant l'exécution α selon un ordre total particulier. Par exemple, un ordre total possible est celui détaillé dans la figure 2.1 (à savoir {a, b, c2 !m, c1 !m, c1 ?m, c2 ?m, b}). Ce type de représentation est appelé un diagramme espace-temps (ou diagramme temporel, ou chronogramme). A chaque processus p_i est associée une ligne continue horizontale modélisant l'écoulement du temps (de la gauche vers la droite). L'instant où un événement e_i^x se produit est matérialisé par un petit tiret vertical sur la ligne de p_i. Si, grâce à un référentiel de temps commun à tous les processus, tous les événements peuvent être estampillés par la date à laquelle

il se sont produits, la construction de ce type de diagramme est possible. Cependant, en raison de l'absence d'une horloge globale commune, cet ordre total ne peut généralement pas être observé.

A défaut de pouvoir raisonner sur un ordre total, il est possible d'analyser le comportement observé au cours de l'exécution α en considérant uniquement l'ordre partiel correspondant à la relation $\prec^\alpha$. Cette information peut être obtenue sans avoir besoin d'une horloge globale.

Au cours d'une exécution, les dépendances causales entre les événements peuvent être capturées en ayant recours à un mécanisme d'horloge logique. Une horloge vectorielle [Fid87, M+88] notée $V(e)$ peut être associée à chaque événement e. Par construction, ce vecteur de taille n est tel que $V(e)[i]$ représente le nombre d'événements exécutés par le processus p_i qui précèdent causalement l'événement e. Le calcul de cette information au vol nécessite de faire transiter dans chaque message, la valeur de l'horloge vectorielle de l'émetteur au moment de l'émission du message. Un mécanisme d'horloge vectorielle capture fidèlement la relation de causalité au sens où :

$$\forall e \in E_\alpha, \forall f \in E_\alpha - \{e\}, \quad e \prec^\alpha f \Longleftrightarrow V(e) < V(f)$$

Si les estampilles des événements n'ont pas été calculées au cours de l'exécution, il est parfois possible de déduire cette information à partir de la trace. Ainsi, si les canaux de communication délivrent leurs messages selon un ordre FIFO (First in, First out), la mise en relation entre un événement de réception et l'événement d'émission qui lui correspond peut facilement être effectuée lors d'une exploration séquentielle des logs des processus. En conséquence, l'ordre partiel entre les événements de E^α peut être déduit au cours d'une seule lecture des n fichiers de logs.

A partir de l'ordre partiel, il est possible de définir des coupures cohérentes [Gar12b]. Une coupure (cohérente ou pas) est un sous-ensemble d'événements.

Définition 3 *C correspond à une coupure si et seulement si $C \subset E^\alpha$ et $\forall e_i^x \in C, x > 1 \Rightarrow e_i^{x-1} \in C$.*

Une coupure découpe la séquence des événements exécutés par chaque processus p_i en deux parties : un passé et un futur. De fait, une coupure identifie pour chaque processus p_i un état local qui est atteint par p_i une fois qu'il a exécuté les événements de la coupure le concernant. Il est donc possible d'associer à une coupure, un état global comportant un état local par processus. Une coupure cohérente correspond à un état global qui peut être atteint pendant le calcul.

Définition 4 *C correspond à une coupure cohérente si et seulement si $C \subset E^\alpha$ et $\forall e \in C, \forall f \in E^\alpha, f \prec^\alpha e \Rightarrow f \in C$.*

Par définition, si un événement appartient à une coupure cohérente, tous ses prédécesseurs (par rapport à la relation d'ordre partiel $\prec^\alpha$) figurent également dans cette coupure. L'ensemble de toutes les coupures cohérentes qui peuvent être définies avec les événements de E^α est noté $\mathcal{C}^\alpha$. Par construction, cet ensemble est inclus dans l'ensemble des parties de E^α. Deux coupures cohérentes particulières, $\emptyset$ et E^α correspondent à l'état initial et l'état final qui seront atteints durant l'exécution α.

La figure 2.2 illustre la notion de coupure. Deux coupures, l'une cohérente et l'autre pas, y sont représentées. Une coupure est matérialisée par une frontière (ligne en pointillés) et tous les événements qui apparaissent à gauche de cette frontière sont des éléments de la coupure. La coupure cohérente est en bleu et correspond aux quatre événements de types $\{a,\ b,\ c2!m,\ c1!m\}$. Chacun de ces quatre événements a l'intégralité de ses prédécesseurs (par rapport à la relation $\prec^\alpha$) inclus dans la coupure bleu. A noter que dans l'état global correspondant, deux messages sont émis mais pas encore reçus (messages en transit). La seconde coupure (en rouge) est incohérente. Elle fait référence aux quatre événements ayant pour types $\{a,\ b,\ c1!m,\ c2?m\}$. Dans cette coupure, l'événement $c2?m$ a quatre prédécesseurs mais l'un d'eux (en l'occurrence $c2!m$) n'appartient pas à la coupure. L'état global identifié par la coupure rouge n'est pas un état qui peut être atteint durant l'exécution car il fait référence à un message reçu qui n'aurait pas été émis.

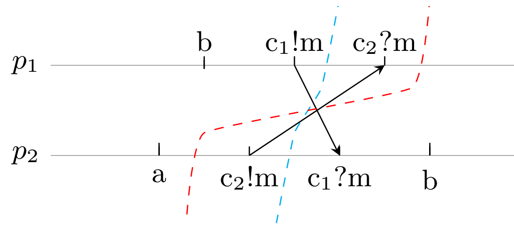

FIGURE 2.2 – Exemples de coupure cohérente (bleu) / incohérente (rouge)

Si C_x et C_y sont deux éléments de $\mathcal{C}^\alpha$, alors $C_x \cup C_y$ et $C_x \cap C_y$ appartiennent également à $\mathcal{C}^\alpha$. L'ensemble des coupures cohérentes $\mathcal{C}^\alpha$ muni de la relation d'inclusion $\subseteq$ a une structure de treillis. En effet d'une part, la relation binaire $\subseteq$ est une relation d'ordre sur $\mathcal{C}^\alpha$ et, d'autre part, toute paire de coupures cohérentes admet à la fois une borne supérieure et une borne inférieure dans $\mathcal{C}^\alpha$. Le treillis $(\mathcal{C}^\alpha, \subseteq)$ qui est composé des coupures cohérentes qui sont classées à l'aide de la relation inclusive $\subseteq$ est représenté

graphiquement dans la figure 2.3. Dans cette représentation, chaque point (rond blanc) correspond à une coupure cohérente.

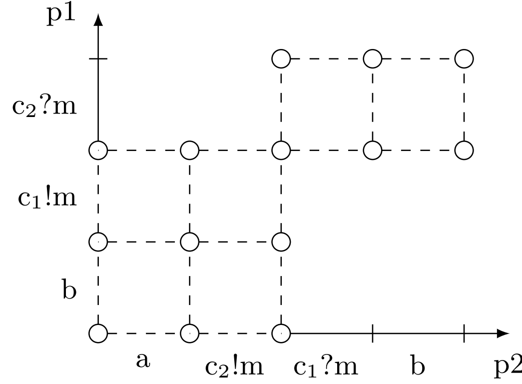

FIGURE 2.3 – Treillis des états globaux cohérents correspondant à l'exécution α

Définition 5 *Soit une exécution α durant laquelle l^α événements sont observés. Un chemin dans le treillis associé est défini par une séquence de $l^\alpha + 1$ coupures cohérentes dénotée (C_0,..., C_k, ..., C_{l^α}) telle que : $C_0 = \emptyset$ et $\forall k$, $0 < k \leq l^\alpha$, $C_k \in \mathcal{C}^\alpha$ et $C_{k-1} \subset C_k$.*

Le long de tout chemin (C_0,..., C_k, ..., C_{l^α}) du treillis, la propriété suivante est vérifiée : $\forall k$, $0 < k \leq l^\alpha$, $\exists e \in E^\alpha$ tel que $C_{k-1} \cup \{e\} = C_k$. De fait, un chemin dans le treillis correspond à un ensemble totalement ordonné de l^α événements.

Dans le treillis représenté dans la figure 2.3, il est possible d'énumérer 18 chemins distincts. Parmi ces chemins, on peut retrouver le chemin correspondant à l'ordre total détaillé dans la figure 2.1 (à savoir {a, b, c2 !m, c1 !m, c1 ?m, c2 ?m, b}). Si ce chemin particulier correspond au comportement réel de l'application, les 17 autres chemins correspondent à des ordres totaux également compatibles avec l'ordre partiel observé. De fait, il s'agit de comportements possibles lors de l'exécution observée. En l'absence d'horloge globale, l'analyse du comportement de l'application peut se faire en considérant l'ensemble des 18 comportements possibles ou uniquement un sous-ensemble de ces chemins. Dans ce dernier cas, rien ne permet de garantir que le comportement réel est l'un des chemins retenus.

2.3 Apprentissage d'un modèle de comportement

Cette section est consacrée à deux approches de modélisation des comportements d'une application distribuée : l'une des approches est fondée sur des automates et l'autre

sur des invariants. Dans un premier temps, les usages de tels modèles et les approches générales de construction sont brièvement décrits. Ensuite, cette section détaille des techniques utilisées pour inférer ces deux types de modèles (automates et invariants) lors d'une phase d'apprentissage.

2.3.1 Contexte

La modélisation d'application est utilisée à différentes fins liées à l'ingénierie logicielle :

- La compréhension des comportements : La modélisation d'une application peut être utilisée pour comprendre et étudier l'impact de changements dans celle-ci [RBDL97]. Le modèle permet d'envisager plus facilement l'implémentation de modifications ;
- La vérification de programmes [HHK+15, LMTY02] : Elle est utilisée pour le développement d'applications critiques. Elle consiste, par exemple, à parcourir le code et à le confronter à un modèle pour prouver certaines propriétés ;
- La détection d'erreurs [SBN+16] : Un modèle peut être utilisé indirectement (génération de test case) ou directement (utilisation du modèle lors de l'exécution de l'application) pour détecter les comportements non spécifiés ou les erreurs de fonctionnement.

La construction de modèles d'applications peut se faire selon deux types d'approches : statique et dynamique. Les approches statiques consistent à construire le modèle en utilisant le code source de l'application ou en spécifiant le fonctionnement de celui-ci. Les travaux de Murphy [MNS01] utilisent ce type d'approche. Ce dernier définit des expressions régulières pour extraire un modèle abstrait à partir du code source. Les approches dynamiques visent à inférer un modèle de comportement à partir d'observations du fonctionnement de l'application. Cette inférence peut se faire notamment en exploitant les contenus des fichiers de logs d'exécution. Des travaux utilisant ce type d'approche seront énoncés dans les sections suivantes.

Dans cette thèse, nous souhaitons mettre en évidence la pertinence de la détection d'erreurs appliquée au domaine de la détection d'intrusion à l'aide d'une approche dynamique.

2.3.2 Inférence d'automates

L'inférence d'automates selon une approche dynamique a fait l'objet de plusieurs études [ABL02, LMP08, LMS12]. Ces travaux utilisent des fichiers de logs d'exécutions

dites positives (exécutions sans erreur) pour construire un automate. La plupart de ces approches utilisent un algorithme provenant du domaine de l'inférence grammaticale [BF72] ou des dérivés de celui-ci [LMP08, PMM17]. L'inférence d'automates pour les systèmes distribués peut se faire selon une approche similaire mais nécessite de prendre en compte les spécificités de ces systèmes (voir section 2.2).

Pour modéliser le comportement d'une application distribuée sous la forme d'un automate, on peut considérer d'utiliser plusieurs types de structures (I/O Automata [Lyn88], CFSM [BZ83], Interface Automata [DAH01], FSM, ...). En lien fort avec cette étude, les articles [BBEK14] et [THH+16] considèrent respectivement des CFSM (de l'anglais "Communicating Finite State Machine") et FSM (de l'anglais "Finite State Machine") lors de la phase d'apprentissage du comportement d'une application. Ils présentent des solutions qui ont un but similaire. La première approche a pour objectif de modéliser le comportement d'un système distribué pour la détection de bugs et la compréhension de son fonctionnement. La seconde approche a pour but la détection d'attaques à l'aide de mécanisme de détection d'erreurs.

Dans [BBEK14], les n CFSMs créés représentent les comportements des n processus pris séparément. Autrement dit, durant une exécution α, pour chaque processus p_i, un automate modélise la séquence locale d'événements E_i^α. Les automates communiquent entre eux à l'aide d'opérations "envoi" et "réception". Dans [THH+16], un FSM unique est créé et correspond à la représentation graphique du treillis des états globaux cohérents. Les comportements acceptés par l'automate correspondent aux chemins du treillis.

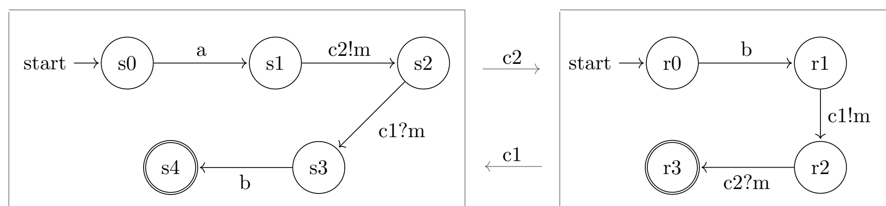

Figure 2.4 – CFSM correspondant à l'exécution α

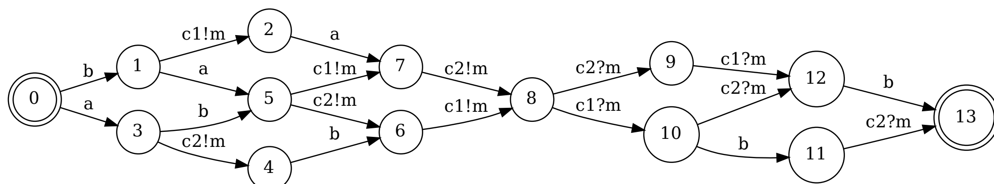

Figure 2.5 – FSM correspondant à l'exécution α

Les figures 2.4 et 2.5 contiennent les deux représentations de l'exécution α qui sont obtenues en appliquant ces deux approches distinctes. Le CFSM (voir figure 2.4) est composé de deux automates (un par processus). Le FSM (voir figure 2.5) correspond à la représentation visuelle du treillis des états globaux cohérents de la figure 2.3. Ces deux automates reconnaissent exactement les mêmes traces. En revanche, ils présentent des caractéristiques différentes comme le nombre d'états et de transitions ou encore la représentation des canaux de communications.

Puisqu'une application peut avoir des comportements corrects différents d'une exécution à une autre, une phase d'apprentissage nécessite de considérer plusieurs exécutions de l'application. Au final, l'automate résultant doit accepter au moins l'ensemble des comportements observés au cours de chacune de ces exécutions. Pour construire cet automate, dans le cas des deux approches que nous venons de citer, les modèles obtenus à partir de chaque exécution sont fusionnés. Dans [BBEK14], cette fusion se fait au travers de la construction d'un graphe orienté acyclique appelé *concrete FSM* qui énumère l'ensemble des chemins d'exécutions possibles entre les différents processus. En combinant le comportement local d'un processus observé durant une exécution avec le comportement local d'un autre processus observé durant une autre exécution, il est cependant possible d'introduire dans le modèle des comportements globaux qui n'ont jamais été vu au cours d'une des exécutions réalisées durant la phase d'apprentissage. Ces comportements qui seront acceptés par le modèle construit peuvent être des comportements possibles de l'application ou, au contraire, des comportements impossibles dans des conditions normales d'exécution. Dans le cas de la seconde approche, dans [THH+16], les automates qui correspondent aux treillis de chaque exécution sont construits dans un premier temps et leurs états initiaux sont ensuite fusionnés pour obtenir un seul automate (pas nécessairement déterministe). Ainsi, le FSM obtenu accepte toutes les séquences apprises et uniquement celles-ci.

Pour illustrer cette phase de fusion, nous considérons à nouveau l'exécution α et nous supposons que l'apprentissage va aussi tenir compte de la deuxième exécution notée β. Les modèles obtenus à l'issu de la phase de fusion sont représentés dans les figures 2.6 et 2.7 pour chacune des deux approches.

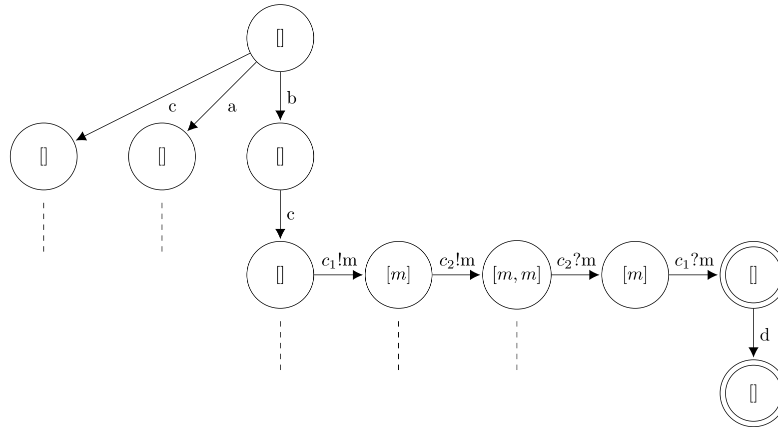

FIGURE 2.6 – CFSM représentant la fusion des exécutions α et β

Le *concrete FSM* (voir figure 2.6) n'est que partiellement décrit car notre objectif est surtout de montrer que les combinaisons d'activités de processus peuvent conduire à des comportements nouveaux qui sont logiques du point de vue des communications mais qui ne correspondent pas à ce qui a été observé, ni durant l'exécution α, ni durant l'exécution β. Sur la figure, les chemins possibles sont partiellement représentés par leur suffixe dont les prolongements possibles sont remplacés par des traits verticaux en pointillés. Pour chaque état de l'automate, sont identifiés les messages en transit sur chaque canal. Plus précisément, à l'intérieur de chaque cercle matérialisant un état figure entre crochets la liste des messages émis et non reçus. L'automate obtenu combine des comportements de processus qui ont été appris. Dans cette figure, l'automate qui n'est que partiellement représenté, fait apparaître deux comportements globaux particuliers résultant d'une part de la combinaison du comportement de p_1 durant l'exécution α et de p_2 durant l'exécution β (à savoir le comportement {b, c, c1 !m, c2 !m, c2 ?m, c1 ?m}) et, d'autre part, de la combinaison du comportement de p_1 durant l'exécution β et de p_2 durant cette même exécution β (à savoir le comportement {b, c, c1 !m, c2 !m, c2 ?m, c1 ?m, d}). La reconnaissance de chacun de ces deux comportements (dont l'un est le préfixe de l'autre) mène à deux états finaux distincts matérialisés par des doubles cercles. Le comportement {b, c, c1 !m, c2 !m, c2 ?m, c1 ?m }, qui ne figure dans aucune des deux traces initiales, correspond à deux comportements locaux possibles pour p_1 et p_2 et à un échange de message cohérent entre ces deux processus. Il est donc considéré comme un comportement acceptable (bien que non appris lors d'une exécution unique). A noter que le comportement nouveau n'est pas toujours le préfixe d'un comportement réellement observable. Ainsi, la composition du comportement de p_1 durant l'exécution β avec le comportement de p_2 durant l'exécution α donne également un comportement acceptable

(non représenté sur la figure) qui n'est pas le préfixe d'un comportement observé au cours d'une seule exécution. Dans l'exemple, les deux exécutions α et β partagent un point commun : sept événements ont été observés durant chacune d'elle. Les deux nouveaux comportements globaux introduits sont quant à eux composés respectivement de 6 et 8 événements. L'introduction de nouveaux comportements globaux fait donc disparaître du modèle des propriétés qui peuvent éventuellement être des caractéristiques essentielles de l'application.

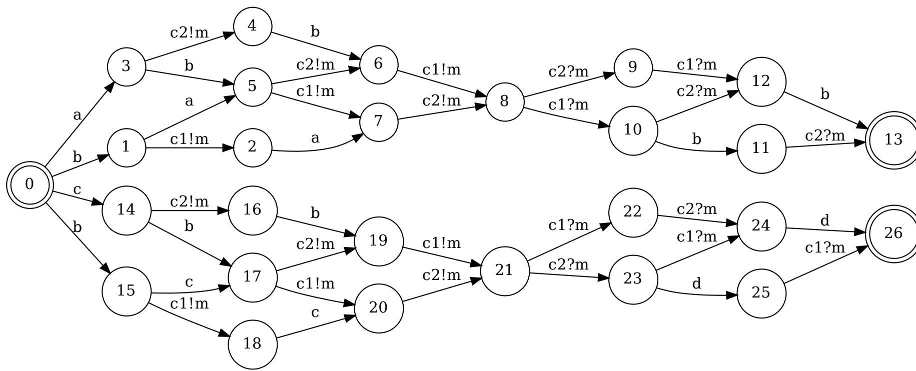

Figure 2.7 – FSM correspondant à la fusion des exécutions α et β

Pour ce qui est de la deuxième approche, le FSM (figure 2.7) correspond à la fusion des états initiaux des automates correspondant aux exécutions α et β obtenus via la seconde approche. Ici contrairement au *concrete FSM*, l'automate complet est fourni afin de constater qu'il n'accepte pas de chemins qui ne sont pas présents dans les traces initiales.

Une fois le FSM ou le *concrete FSM* obtenu, les deux approches sont suivies ensuite d'une phase de raffinement de l'automate obtenu aussi appelée phase de généralisation. L'approche présentée dans [BBEK14] vise à construire un CFSM à l'aide du concrete FSM. Pour cela, certains états seront fusionnés à condition que ces fusions ne violent pas des propriétés préalablement identifiées. Des invariants (voir la section suivante) sont donc utilisés non pas pour avoir un second modèle mais pour contrôler la transformation du modèle à base d'automates. Dans l'approche présentée en [THH+16], les états ayant des futurs similaires vont être fusionnés à l'aide d'un algorithme de généralisation [BF72].

2.3.3 Inférence d'invariants

Un comportement observé durant une exécution peut également être caractérisé par une liste de propriétés qui ont été vérifiées durant cette exécution. Ces propriétés peuvent être variées et porter notamment sur l'état des processus (valeurs des variables) ou sur l'enchaînement des événements. Lorsque que la même propriété est satisfaite au cours de toutes les exécutions observées durant la phase d'apprentissage, la propriété est un invariant ou, plus exactement, un invariant probable si rien ne permet d'affirmer que toute autre exécution non apprise va aussi satisfaire la propriété. Cette section traite des différents types d'invariants, de leurs utilisations possibles et des façons de les inférer. Tout comme les automates, ils peuvent être inférés statiquement ou dynamiquement et peuvent être utilisés pour différents objectifs (voir la section 2.3.1). Cette thèse se concentre sur les invariants temporels décrivant le comportement d'applications distribuées. Cependant, d'autre types d'invariants peuvent aussi être envisagés [ECGN01, Dev00] pour inférer des modèles.

Cette section traite du passage de définitions de propriétés en logique temporelle à l'inférence automatique d'invariants pour les applications réparties. Elle est divisée en quatre sous-sections qui traitent respectivement des propriétés de Logique temporelle, de la simplification de ces propriétés, de l'inférence automatique de propriétés et enfin de l'inférence de ce propriétés pour les applications réparties.

2.3.3.1 Logique temporelle

La notion d'invariants temporels apparaît en 1977 avec [Pnu77] qui propose une logique temporelle linéaire (LTL) pour effectuer une vérification formelle des programmes. Une propriété s'exprime sous la forme d'une expression logique permettant de raisonner à la fois sur les étapes de calculs (via des propositions et des opérateurs logiques comme « $\vee$ », « $\wedge$ » et « $\rightarrow$ ») et leur enchaînement (via les opérateurs temporels modaux comme « $\bigcirc$ » appelé Suivant, « $\Box$ » appelé Globalement et « $\Diamond$ » appelé Finalement). Si Φ est une propriété alors :

$\bigcirc\Phi$ signifie que la propriété Φ doit être satisfaite dans l'état suivant ;

$\Box\Phi$ signifie que la propriété Φ doit être satisfaite dans tous les états suivants ;

$\Diamond\Phi$ signifie que la propriété Φ doit être satisfaite dans un des états suivants.

Les aspects temporelles qui caractérisent un programme correct qu'il soit séquentiel ou distribué peuvent ainsi être identifiés et les analyses menées peuvent faire référence au comportement de l'application au fil du temps. Des propriétés de vivacité (quelque

chose de bon finit par se produire) et de sûreté (quelque chose de mauvais de ne produit jamais) peuvent ainsi être spécifiées.

La logique temporelle linéaire [Pnu86] ou des variantes comme la « computation tree logic » (CTL) [CE81] reposent sur des concepts similaires. Les propositions formulées à l'aide de ces logiques permettent de modéliser de nombreuses propriétés de l'application. Mais ces propriétés peuvent être complexes dans leur spécifications et difficiles à inférer de manière automatique.

2.3.3.2 Simplification des propriétés

En 1999, Dwyer [DAC99] propose de se focaliser sur un ensemble réduit de propriétés afin de simplifier la phase de spécification. L'idée est d'identifier quelques classes de propriétés relativement simples. Chaque propriété ne mentionne qu'un nombre limité d'étapes de calcul : le plus souvent, deux seulement. Une étape de calcul identifiée dans une propriété est caractérisée par le fait qu'un événement particulier se produit ou que l'état atteint suite à l'exécution de l'événement présente une caractéristique particulière. La définition d'une propriété fait donc référence à l'apparition d'un (ou plusieurs) état(s)/événement(s) particulier (s) durant le calcul. Nous utiliserons ici les lettres majuscules F et G pour faire référence à ces apparitions (qui peuvent par exemple correspondre respectivement à des apparition d'événements de type f et g).

L'article de Dwyer définit également une notion de fenêtre temporelle qui permet d'adapter la portée de l'analyse. La satisfaction d'une propriété peut être évaluée sur l'intégralité de l'exécution (Global) ou sur une ou plusieurs portion(s) du calcul dont le début et la fin peuvent être définis dynamiquement en faisant à nouveau référence à l'apparition d'état/événement (Avant la première apparition F, Après la première apparition F, Entre une apparition F et une apparition G, Après une apparition F et Jusqu'à une possible apparition G ou la fin du calcul). Pour une fenêtre temporelle donnée, les propriétés suivantes qui portent donc sur l'occurrence et l'ordre des apparitions d'état(s)/événement(s) sont proposées :

— **Absence** : Un état/événement n'apparaît pas dans une fenêtre temporelle.

— **Existence** : Un état/événement doit apparaître dans une fenêtre temporelle.

— **Universalité** : Un état/événement doit apparaître tout au long d'une fenêtre temporelle.

— **Existence Bornée** : Un état/événement doit apparaître k fois dans une fenêtre temporelle. Deux autres variantes de cet opérateur spécifiant au moins k occurrences et au plus k occurrences de l'état/événement sont également considérées.

— **Précédence** : Un état/événement F doit toujours être précédé par un autre état/événement G dans une fenêtre temporelle.

— **Réponse** : Un état/événement F doit toujours être suivi par un autre état/événement G dans une fenêtre temporelle.

— **Précédence Chaînée** : Une séquence d'états/événements F_1, ..., F_n doit toujours être précédée par une séquence d'états/événements G_1, ..., G_n dans une fenêtre temporelle.

— **Réponse Chaînée** : Une séquence d'états/événements F_1, ..., F_n doit toujours être suivie par une séquence d'états/événements G_1, ..., G_n dans une fenêtre temporelle.

Pour savoir si le fait de contraindre le format des propriétés qui pouvaient être employées a un impact modéré ou important sur la puissance d'expressivité, une phase d'évaluation a été menée. En s'appuyant sur des spécifications en logique temporelle déjà existantes, un ensemble de 555 propriétés ont été collectées et, pour chacune, Dwyer a tenté de les exprimer à nouveau à l'aide des quelques opérateurs proposés. Parmi l'ensemble initial, 92% des propriétés a pu être reformulé ce qui démontre que la limitation à des propriétés au format simple et pré-défini permet de répondre à la plupart des besoins d'une personne chargée d'établir une spécification.

Comme le montre l'évaluation menée par Dwyer, des équivalences entre les formulations de propriétés proposées dans différents travaux peuvent être trouvées. Pour illustrer ces réécritures possibles, nous considérons à nouveau l'exemple proposé dans la section 2.2 et nous considérons une fenêtre temporelle correspondant à l'intégralité des calculs. Durant les exécutions α et β, l'une des propriétés satisfaites durant les deux exécutions peut s'exprimer en langage naturel sous la forme suivante : « Un événement de type a est toujours suivi par un événement de type b ». A noter que lors de l'exécution β, l'absence d'événement de type a fait que la propriété est nécessairement satisfaite. Cette propriété reformulée avec des opérateurs LTL donne l'expression $\square(\mathrm{a} \rightarrow \lozenge\ \mathrm{b})$ où l'opérateur Globalement noté « $\square$ » indique que la propriété énoncée entre parenthèse est vrai à toutes les étapes de calcul, l'opérateur logique « $\rightarrow$ » correspond à la notion classique d'implication et l'opérateur Finalement « $\lozenge$ » précise que la propriété énoncée va nécessairement être vrai à un moment donné dans le futur. L'opérateur représentant cette propriété dans l'article de Dwyer serait « Réponse ».

A noter que les propriétés de type « Précédence » et « Réponse » identifient des dépendances temporelles (potentiellement des causalités) entre les événements à l'intérieur d'une fenêtre temporelle : l'apparition d'un événement est conditionné par l'apparition d'un autre. La vérification de ces propriétés nécessite de pouvoir raisonner sur la séquence d'événements qui s'est produit durant la fenêtre temporelle (connaissance d'un

ordre total sur les événements) ou sur l'ensemble des séquences qui ont pu se produire durant cette fenêtre (connaissance d'un ordre partiel sur les événements). Ainsi, dans l'exemple de la propriété « Un événement de type a est toujours suivi par un événement de type b », durant l'exécution α où seules les dépendances causales ont été observées, tous les ordres totaux compatibles avec la relation d'ordre partiel $\prec^{\alpha}$ correspondent à des séquences d'événements où le premier événement de p_2 (de type a) doit nécessairement être suivi du dernier événement de p_2 (de type b) pour que la propriété soit satisfaite.

2.3.3.3 Inférence automatique de propriétés

Pour construire une liste de propriétés temporelles qui vont caractériser une exécution d'un programme, il faut tout d'abord choisir et décrire des propriétés présentant un intérêt, puis s'assurer qu'elles sont satisfaites durant l'exécution. Cette connaissance peut provenir d'une spécification déjà existante de l'application ou être déterminée manuellement pour de petits ensembles de propriétés. Dans le cas où le nombre de propriétés devient important, évaluer chacune d'elles pour valider le fait qu'elle est satisfaite nécessite d'avoir recours à un processus automatisé. Des travaux ont proposé d'inférer des invariants (ou propriétés) probables à l'aide de traces d'exécutions [YEB+06, LFY+10, BABE11, THH+16]. Les opérateurs utilisés pour définir des invariants probables correspondent généralement aux opérateurs précédemment décrits (Absence, Réponse et Précédence) avec cependant quelques différences. La fenêtre temporelle envisagée pour les deux premiers opérateurs correspond à l'ensemble de l'exécution (fenêtre Globale) et aux fenêtres Avant et Après pour le troisième et dernier opérateur. Les trois opérateurs mis en avant dans ces travaux sont aussi ceux qui sont les plus fréquemment utilisés dans l'approche de Dwyer. Les différences entre les approches utilisant ces opérateurs [LFY+10, BABE11, THH+16] sont liées à la sémantiques des opérateurs ou encore à l'apparition d'opérateurs additionnels.

Dans l'article [LFY+10], deux variantes sémantiques des opérateurs « Précédence » et « Réponses » sont présentes alors que l'opérateur « Absence » n'est pas considéré. On retrouve les opérateurs suivants :

— « Dépendance Future » (DF) : L'occurrence d'un événement de type a est suivie de l'occurrence d'au moins un événement de type b (noté a $\rightarrow_f$ b).

— « Dépendance Future Stricte » (DFS) : Chaque occurrence d'un événement de type a est suivie de l'occurrence d'au moins un événement associé de type b (noté a $\rightarrow_{sf}$ b).

— « Dépendance Passée » (DP) : L'occurrence d'un événement de type b est précédé de l'occurrence d'au moins un événement de type a (noté a $\rightarrow_b$ b).

— « Dépendance Passée Stricte » (DPS) : Chaque occurrence d'un événement de type b est précédé de l'occurrence d'au moins un événement associé de type a (noté a $\rightarrow_{sb}$ b).

Parmi ces quatre types d'opérateur qui s'appliquent ici sur la globalité du calcul, DF et DP se concentrent sur la relation temporelle entre deux types d'événements, tandis que DFS et DPS regardent non seulement la relation temporelle mais aussi le nombre d'occurrences des types d'événements. Ainsi, la propriété a $\rightarrow_f$ b est satisfaite par une séquence d'événements si toutes les occurrences d'un événement de type a dans cette séquence sont suivies par un événement de type b (sans que les occurrences d'événements de type b soient nécessairement distinctes). Par exemple, cette relation est satisfaite par les séquences {a, a, b, b}, {a, b, a, b}, {a, b, b, b}, {b, a, b, b}, {b, b, a, b}, {b, b, b, b}, {a, a, a, b} et {b, a, a, b}. A l'inverse, pour que la relation a $\rightarrow_{sf}$ b soit satisfaite, il faut pouvoir associer (en respectant la contrainte temporelle) des occurrences d'événements de type b distinctes à chacune des occurrences d'événements de type a. De faite, cela implique en particulier que le nombre d'occurrences d'événements de type b soit supérieur ou égal au nombre d'occurrences d'événements de type a. Cette relation est donc satisfaite par les séquences {a, a, b, b}, {a, b, a, b}, {a, b, b, b}, {b, a, b, b}, {b, b, a, b} et {b, b, b, b} mais elle n'est pas satisfaite par les séquences {a, a, a, b} et {b, a, a, b}. Bien évidement, les invariants « a $\rightarrow_{sf}$ b » et « a $\rightarrow_{sb}$ b » impliquent respectivement les invariants « a $\rightarrow_f$ b » et « a $\rightarrow_b$ b ».

L'article [BABE11] utilise la sémantique des opérateurs DFS et DPS pour ses opérateurs « Précédence » et « Réponses ». Il définit aussi l'opérateur « Absence » et des opérateurs liés à la concurrence.

L'article [THH+16] reprend la sémantique des opérateurs DF et DP dans son approche et considère l'opérateur « Absence ». Les classes de propriétés « toujours suivi par », « toujours précédé de » et « jamais suivi de » qui sont utilisées dans l'approche [THH+16] sont également considérées dans le cadre de cette thèse. Nous allons donc les définir formellement en nous appuyant sur les notions d'événements et de type d'événement.

Par rapport à une exécution α, si a et b sont deux types d'événements observés durant cette exécution (*i.e.*, $a \in \overline{E^\alpha}$ et $b \in \overline{E^\alpha}$), les trois classes de propriétés citées seront notées de la façon suivante :

— $a \rightarrow^\alpha b$: durant l'exécution α, un événement de type a est toujours suivi par un événement de type b.

— $a \leftarrow^\alpha b$: durant l'exécution α, un événement de type b est toujours précédé par un événement de type a.

— $a \nrightarrow^{\alpha} b$: durant l'exécution α, un événement de type a n'est jamais suivi par un événement de type b.

2.3.3.4 Inférence de propriétés pour les applications réparties

Lorsque l'exécution d'une application répartie α est observée sous la forme d'un ordre partiel, un événement e est toujours suivi d'un événement f (respectivement, toujours précédé d'un événement f) dès lors que $e \prec^{\alpha} f$ (respectivement, $f \prec^{\alpha} e$). Inversement, dès lors que deux événements e et f sont concurrents, il existe au moins un chemin du treillis où e apparaît avant f et au moins un chemin où f apparaît avant e. Partant de ce constat, nous pouvons donc formellement définir les trois classes de propriétés portant sur les types d'événement de la façon suivante :

Définition 6 *Classe de propriétés « toujours suivi par » :*
$\forall (a, b) \in \overline{E^{\alpha}}^2$, *$a \rightarrow^{\alpha} b$ si et seulement si* $\forall e \in E^{\alpha}(a), \exists f \in E^{\alpha}(b)$ *tel que* $e \prec^{\alpha} f$

Définition 7 *Classe de propriétés « précède toujours » :*
$\forall (a, b) \in \overline{E^{\alpha}}^2$, *$a \leftarrow^{\alpha} b$ si et seulement si* $\forall f \in E^{\alpha}(b), \exists e \in E^{\alpha}(a)$ *tel que* $e \prec^{\alpha} f$

Définition 8 *Classe de propriétés « jamais suivi par » :*
$\forall (a, b) \in \overline{E^{\alpha}}^2$, *$a \nrightarrow^{\alpha} b$ si et seulement si* $\forall e \in E^{\alpha}(a), \nexists f \in E^{\alpha}(b)$ *tel que* $(e \prec^{\alpha} f) \vee (e \parallel f)$

Si $x > 0$ est le nombre de types d'événement distincts observés durant l'exécution α (*i.e.* $x = \mid \overline{E^{\alpha}} \mid$), la liste des propriétés satisfaites par cette exécution identifie entre 0 et $3x^2$ propriétés. En effet, il y a 3 classes et, pour chacune, x^2 couples de type d'événement différents sont envisageables. Sachant que, pour tout type d'événement a, les propriétés particulières $a \rightarrow^{\alpha} a$ et $a \leftarrow^{\alpha} a$ sont nécessairement fausses, le nombre maximal de propriétés contenus dans cette liste est en fait borné plus finement par $3x^2 - 2x$.

Par extension, nous utilisons les notations $a \rightarrow b$, $a \leftarrow b$ et $a \nrightarrow b$ (sans faire référence à une exécution particulière) lorsque la propriété est un invariant probable : la propriété a été satisfaite lors de toutes les exécutions apprises.

Pour l'exemple présenté dans la section 2.2, on obtiendrait une liste d'invariants probables comportant le sous-ensemble d'invariants : $a \rightarrow b$, $c \leftarrow f$, $c2?m \nrightarrow c2!m$.

2.4 Conformité d'un modèle avec les comportements réels

La conformité d'un modèle de comportement d'une application par rapport aux comportements réels possibles de cette application peut se mesurer selon deux facteurs : la complétude et la précision. Un modèle de comportement est considéré être complet s'il couvre l'ensemble des comportements possibles de l'application. Il est considéré être précis s'il n'accepte aucun comportement qui ne soit pas un comportement possible de l'application. Un modèle idéal est donc à la fois complet et précis mais ces niveaux de complétude et de précision sont rarement atteints. Ces deux facteurs ainsi que les méthodes permettant de générer le modèle peuvent être évalués lors d'une phase de validation croisée [AC+10].

Ces problématiques font l'objet de travaux depuis de nombreuses années au sein de diverses communautés. En particulier, le problème de la construction d'un modèle a des liens forts avec la théorie des langages. Durant une exécution, le ou les comportements observés correspondent en effet à des séquences d'événements typés. Identifier des comportements acceptables revient en quelque sorte à définir un langage où l'alphabet correspond à l'ensemble des types d'événements possibles et les mots sont les séquences de types qu'une exécution correcte peut produire. L'inférence grammaticale [DlH10] a pour objectif d'apprendre, à partir de quelques informations sur un langage, une grammaire inconnue qui caractérise ce langage. La phase d'apprentissage, qui permet de construire un modèle à partir de quelques mots du langage cible, entre parfaitement dans ce cadre.

2.4.1 Complétude du modèle

Durant une phase d'apprentissage, un nombre fini de comportements sont observés. De fait, une phase d'apprentissage prenant en compte un nombre insuffisant d'exécutions peut impacter négativement la complétude du modèle. Par définition, le modèle construit rejette les comportements qui ne satisfont pas les contraintes identifiées. Moins un modèle est contraignant, plus sa complétude a des chances d'être élevée. Ainsi, dans le cas d'un modèle à base d'invariants, la complétude peut éventuellement être améliorée en réduisant la liste des propriétés qui caractérisent l'application tandis que, dans un modèle à base d'automate, une amélioration peut être obtenue en transformant l'automate afin qu'il accepte un sur-langage. Mais, bien évidement, la difficulté majeure est d'améliorer la complétude sans totalement sacrifier la précision.

L'inférence d'un modèle de comportement d'une application à partir de traces d'exécutions positives est calculable [BB75]. Cependant, inférer l'ensemble des comportements possibles selon cette méthode est un problème NP-complet [Gol67]. Pour obtenir un modèle de comportement acceptant cet ensemble de comportements en temps polynomial, il

est possible d'approximer le modèle à l'aide d'algorithmes [PW93]. Les algorithmes d'approximation vont chercher à construire un automate minimal en réduisant la taille du modèle. Dans [CW98a], les auteurs proposent une comparaison entre plusieurs approches (RNET[DM94], k-tail[BF72], Markov[MQ88]) qui visent à approximer un automate minimal complet.

Dans cette thèse nous nous intéressons plus particulièrement à l'algorithme k-tail. Cet algorithme prend en paramètres d'entrée un automate et un entier positif non nul k. Il produit une version réduite de l'automate initial en fusionnant les états qui partagent les mêmes k-futurs (mêmes séquences futures de longueur k). Deux états (qui ne sont pas forcement atteints après une reconnaissance d'un même préfixe) sont donc compatibles et confondus si toute extension de k symboles qui peut être acceptée à partir de l'un des deux états peut également l'être à partir de l'autre. Ces fusions d'états ont pour conséquence l'introduction de nouveaux comportements dont certains sont corrects bien que n'ayant pas été appris. L'utilisation de l'algorithme k-tail peut donc contribuer à une amélioration de la complétude. En particulier, les automates d'états finis obtenus à partir d'un treillis (voir section 2.3.2) ne comportent aucun circuit et accepte de fait un nombre fini de comportements. Le recours à l'algorithme k-tail permet d'obtenir un nouvel automate qui comporte souvent des boucles et des circuits et qui accepte donc un nombre infini de comportements. Ceci permet de considérer durant la phase d'apprentissage des exécutions dont la durée est bornée tout en pouvant utiliser efficacement les modèles construits pour reconnaître ensuite des comportements plus longs ou infinis qui n'ont pas été observés durant la phase d'apprentissage.

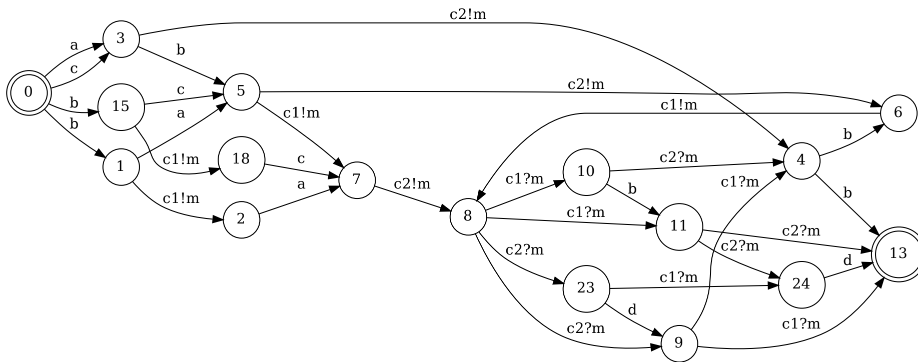

FIGURE 2.8 – Généralisation de l'automate (figure 2.7) avec k=1

Considérons à nouveau les exécutions α et β et appliquons l'algorithme k-tail en

prenant en paramètre d'entrée l'automate qui a été obtenu en fusionnant les deux automates correspondant à ces deux exécutions (voir figure 2.7). L'utilisation de l'algorithme k-tail avec un k=1 produit l'automate représenté dans la figure 2.8. L'algorithme k-tail a fusionné les états partageant le même k-futur de longueur 1. Par exemple les états 4, 12 et 16 qui partageaient la même séquence future <b> sont représentés par l'état 4 dans l'automate. Les fusions ont réduit le nombre d'états de 27 à 17 et ont permis d'introduire des boucles ainsi que de nouvelles séquences qui peuvent être soit correctes soit incorrectes. Par exemple, la séquence d'événement <a, b, c1 !m, c2 !m, c1 ?m, c2 ?m, b, c1 !m, c1 ?m, b, c2 ?m, d> n'était pas présente dans l'automate d'origine.

Dans cette thèse, nous considérons qu'une liste d'invariants probables constitue un second type de modèle. Chaque invariant définit une contrainte temporelle forte mais plusieurs séquences peuvent satisfaire cette contrainte. Par exemple, l'invariant « a → b » (défini dans la section précédente) peut accepter des séquences tel que <a, a, a, a, b>, <a, b> ou encore <a, b, b, b>. Ce type de modèle ne nécessite donc pas nécessairement une phase de généralisation dont le but serait de permettre l'acceptation de plus de comportements. Néanmoins, comme nous l'avons mentionné précédemment, le fait de diminuer la taille de la liste d'invariants (moins de classe de propriétés, moins de type d'événements pris en compte, ...) joue sur la complétude du modèle.

2.4.2 Précision du modèle

L'algorithme k-tail est connu pour produire des modèles précis pour des applications simples mais quand la complexité de l'application augmente, la précision des modèles inférés se trouve fortement réduite [LK06]. Différentes approches de l'état de l'art essayent de traiter le problème de la perte de précision tout en continuant d'utiliser l'algorithme k-tail. Certaines de ces approches proposent des algorithmes de généralisation basés sur k-tail. On retrouve notamment des algorithmes comme Gk-tail [LMP08] ou Timed k-tail [PMM17]. L'algorithme Timed k-tail va par exemple utiliser la contrainte temporelle présente dans les automates temporisés pour généraliser et améliorer la précision.

D'autres approches utilisent des invariants [BBEK14] ou des métriques (statistique, nombre, fréquence, régularité) [CW98b] pour limiter l'introduction de comportements incorrects. Par exemple, dans [BBEK14] les auteurs vont fusionner les états ayant les même k-futurs si et seulement si les nouvelles séquences introduites par cette fusion ne violent pas la liste d'invariants précédemment inférée.

Enfin, il est possible d'appliquer une approche comme celle décrite dans [THH+16]. Dans cet article, les modèles automate et liste d'invariants sont tout deux conservés. L'automate sera généralisé sans contraintes comme l'automate présenté dans la figure 2.8. Les invariants interviendront lors de la phase de reconnaissance en tant que modèle

complémentaire. Dans l'automate de la section précédente, certaines séquences comme <a, b, c1 !m, c2 !m, c1 ?m, c2 ?m, b, c1 !m, c1 ?m, b, c2 ?m, f> ont été introduites lors du processus de généralisation. Cette séquence sera donc acceptée par l'automate. En revanche, si l'invariant c ← f est présent dans la liste d'invariants calculée, la séquence ne pourra être acceptée.

2.5 Conclusion

Ce chapitre a présenté un état de l'art de la modélisation des applications réparties pour la détection d'intrusion. Les différentes parties de ce sujet sont en lien avec plusieurs domaines. L'état de l'art est divisé en quatre parties. Une première partie a présenté des notions générales sur la détection d'intrusion pour positionner les travaux réalisés lors de cette thèse. Une deuxième partie a été consacrée aux caractéristiques des applications distribués et à leurs représentations. Une troisième partie a abordé la modélisation de comportements à l'aide de traces d'exécution. Enfin une quatrième et dernière partie a porté sur la conformité des modèles de comportements inférés par rapport aux comportements des applications.

Chapitre 3

Une première approche pour la construction d'un modèle à partir de traces

Le chapitre précédent a dressé un panorama étayé d'exemples sur les techniques de détection d'intrusion dans les applications distribuées avec une focalisation sur l'apprentissage de modèles de comportements. L'objectif était de donner un aperçu général des approches et des concepts qui ont été proposés jusqu'à présent. Ce nouveau chapitre se focalise sur la proposition et l'étude d'une première solution complète. Cette solution, qui compose et adapte des propositions déjà formulées, est le résultat de choix qui portent sur les approches suivies, l'architecture modulaire du système de détection et les algorithmes mis en œuvre. Cette solution qui a été conçue et implémentée dans le cadre de cette thèse est une première contribution qui nous permet d'identifier des faiblesses et nous servira ensuite de point de référence lorsque nous proposerons des améliorations pour pallier les problèmes rencontrés.

3.1 Présentation Générale

La solution présentée est fortement inspirée par les travaux de [BBEK14, THH+16]. Comme indiqué dans le chapitre précédent, définir une solution permettant de détecter des intrusions via une approche comportementale consiste d'une part, à spécifier comment le modèle du comportement normal de l'application est construit (phase d'apprentissage) et, d'autre part, à préciser comment ce modèle est ensuite exploité afin de déterminer durant une exécution si le comportement de l'application monitorée est correct ou pas (phase de détection).

Ces deux phases ("phase de construction" et "phase de détection"), qui respectivement produisent et exploitent un modèle de comportement, sont représentées dans la figure 3.1 par deux zones délimitées par des pointillés. Le modèle construit (appelé modèle global) est le seul lien entre ces deux phases.

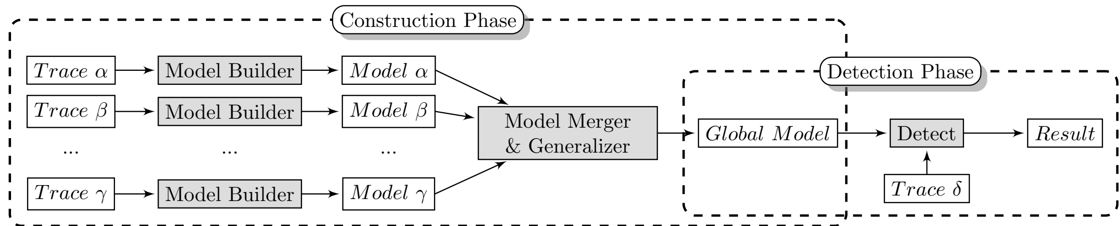

FIGURE 3.1 – Vue générale de l'architecture de l'approche

Afin de donner une vision générale de l'architecture de la solution adoptée, la figure 3.1 identifie les principaux modules logiciels développés. La phase de construction est composée de deux modules *ModelBuilder* et *ModelMerger&Generalizer* dont les principes généraux sont assez similaires à ceux des mécanismes décrits dans [BBEK14, THH+16]. La phase de détection ne comporte que le module *Detect* dont le fonctionnement s'inspire de l'approche [THH+16]. Tout comme dans [THH+16], le modèle est dual au sens où il est composé de deux sous-modèles distinct : d'une part, une représentation des comportements corrects sous la forme d'un automate et, d'autre part, une caractérisation de ces comportements attendus sous la forme d'une liste d'invariants devant être respectés. Nous verrons par la suite (au chapitre 6) qu'il est envisageable d'aller au delà de deux modèles.

Puisque l'apprentissage se fait en exploitant plusieurs traces qui ont été obtenues durant des exécutions distinctes, les approches citées s'appuient sur la notion de modèle intermédiaire. Par définition, un modèle intermédiaire est construit à partir d'une seule et unique trace d'exécution. Il y a donc autant de modèles intermédiaires que de traces utilisées pour l'apprentissage du modèle. Tout comme le modèle global, chaque modèle intermédiaire est dual : il comporte un automate et une liste d'invariants. Le premier module *ModelBuilder* a pour rôle de construire un modèle intermédiaire à partir d'une trace d'exécution. Il est donc exécuté autant de fois qu'il y a de traces apprises. Le deuxième module *ModelMerger&Generalizer* est en charge de construire un modèle global de l'application à partir des différents modèles intermédiaires précédemment construits. Pour cela, les automates intermédiaires et les invariants intermédiaires vont être fusionnés et généralisés. Enfin, lors de la phase de détection, le module *Detect* utilise le modèle de comportement normal pour évaluer une nouvelle trace d'exécution δ. Cette trace est considérée être légitime tant qu'elle est conforme au modèle. Sinon,

une attaque est supposée se dérouler et expliquer la déviation observée par rapport aux comportements attendus : une alerte est alors levée.

Les trois sections suivantes détaillent le fonctionnement des trois différents modules. Des précisions sur leurs mises en œuvre ainsi qu'une première analyse de leurs limitations sont fournis.

3.2 Inférence d'un modèle intermédiaire à partir d'une seule trace

Puisque les comportements corrects d'une application distribuée sont multiples, il est nécessaire de considérer plusieurs exécutions durant la phase de construction du modèle afin de capter une plus grande variété de comportements légitimes sans pour autant être exhaustif. L'application est donc exécutée un nombre fini de fois (exécutions α, β, ..., γ). Chacune de ces exécutions génère une trace (E^α, E^β, ..., E^γ). L'objectif étant de ne capturer que des comportements légitimes, toutes ces exécutions doivent se dérouler sans qu'aucune attaque ne les perturbe. Sans cette hypothèse (qui peut parfois être difficile à assurer dans des systèmes ouverts), le modèle construit va accepter des comportements perturbés par des attaques menées durant la phase d'apprentissage. En conséquence, lors de la phase de détection, aucune alerte ne sera levée lorsque ces mêmes attaques se reproduiront.

Le module *Model Builder* (voir Fig. 3.1) analyse chaque trace séparément et génère un modèle intermédiaire pour chaque exécution. Le fonctionnement de ce module va être détaillé dans la suite de cette section. Sans perte de généralité, la description va faire référence à la construction du modèle intermédiaire correspondant à l'exécution α. Pour rappel, cette exécution a permis la collecte d'une trace E^α structurée en n logs E_i^α. L'ensemble des événements qui ont été observés est partiellement ordonné par la relation d'ordre $\prec^\alpha$. Puisque les modèles construits (qu'ils soient intermédiaires ou globaux) sont tous dual (i.e. composé chacun d'un automate et d'une liste d'invariants), le processus d'inférence d'un modèle intermédiaire va être détaillé en deux temps. Une première section présente le processus d'inférence d'un automate via la construction du treillis des états globaux. La section suivante traite du processus d'inférence des invariants qui repose lui aussi sur une exploitation du même treillis.

3.2.1 Construction du treillis des états globaux cohérents

La trace collectée durant une exécution permet d'identifier les événements qui se sont produits durant cette exécution et la relation d'ordre partiel qui les ordonne. Une structure telle que le treillis des états globaux permet de raisonner sur l'ensemble des

ordres totaux compatibles avec l'ordre partiel observé : chaque chemin de ce treillis qui relie le plus petit élément (état global initial) au plus grand élément (état global final) correspond à un ordre total sur l'ensemble des événements et donc à une séquence légitime d'événements. Déduire un automate à partir du treillis est donc trivial. La difficulté est de construire dans un premier temps le treillis des états globaux cohérents à partir de la trace d'exécution. Le processus de construction suivi est détaillé dans l'algorithme 1.

Algorithm 1 algorithme de construction du treillis

```
1:  function LATTICECONSTRUCT(state, trace)
2:      for p_i in {p_1, p_2, ..., p_n} do
3:          event ← trace.getEvent(state, p_i)
4:          if canEvolve(state, event) then
5:              newstate ← State(state, p_i, event)
6:              newtrans ← Transition(state, newstate, event)
7:              updateMinMax(newtrans)
8:              transitionset.add(newtrans)
9:              if stateset.add(newstate) then
10:                 LATTICECONSTRUCT(newstate, trace − {event})
11:             end if
12:         end if
13:     end for
14: end function
```

L'objectif de cet algorithme est de construire deux ensembles : un ensemble d'états (noté *stateset*) et un ensemble de transitions entre états (noté *transitionset*). Pour rappel, dans une représentation graphique du treillis (voir par exemple la figure 2.3), un état est matérialisé par un point et une transition correspond à un segment de droite reliant deux points. Puisque nous construisons le treillis des coupures cohérentes, chaque état correspond à une coupure cohérente respectant les contraintes énoncées à la section 2.2 (voir définition 4). Sur le plan algorithmique, un état est identifié par ses coordonnées dans le treillis qui définissent une frontière dans la trace entre les événements déjà interprétés et les événements qui restent à consommer. Ainsi, l'état de coordonnées $(x, y, \cdots, z)$ est obtenu après exécution de $x + y + \cdots + z$ évènements correspondant aux x premiers éléments de E_1^α, aux y premiers éléments de E_2^α et ainsi de suite. A chaque état est également associé un ensemble de messages qui ont été émis mais pas reçus : ces messages sont en transit (ou éventuellement perdus). Un état s est donc associé à une paire $< D, M >$ où D est le vecteur de n coordonnées associé à l'état s et M est l'ensemble des messages en transit au moment où l'état s est atteint.

La fonction LATTICECONSTRUCT$(state, trace)$ est une fonction récursive qui prend en entrée deux paramètres : un état global du treillis à partir duquel la construction va progresser et une trace d'exécution en cours de consommation. Lors de l'appel initial, les paramètres fournis correspondent respectivement à l'état global cohérent initial et à la trace complète E^α composée de n fichiers de logs. L'état initial est caractérisé par le fait qu'aucun événement n'a encore été exécuté. Ses n coordonnées sont donc toutes nulles et aucun message n'est en transit. Le premier appel de la fonction a donc la syntaxe suivante : LATTICECONSTRUCT$(< [0, 0, \ldots, 0], \emptyset >, E^\alpha)$. Lors d'un appel de la fonction LATTICECONSTRUCT$()$, partant d'un état courant, une progression du calcul (via l'exécution d'un nouvel événement) est envisagée pour chacun des processus p_i (ligne 2). L'événement dont l'exécution est envisagée est récupéré et sa description est stockée dans la variable *event* (ligne 3). Cet événement qui est le prochain événement devant être exécuté par p_i peut ne pouvoir être exécuté à partir de l'état courant si d'autres événements qui le précèdent au sens de la relation de causalité $\prec^\alpha$ n'ont pas encore été exécutés. C'est le rôle de la fonction *canEvolve* (appelée à la ligne 4) de tester si l'événement peut être exécuté ou pas. Une mise en œuvre possible de cette fonction sera détaillée dans l'algorithme 2. S'il est possible de construire un nouvel état cohérent, cet état est calculé (ligne 5). La transition entre l'état courant et cet état est crée (ligne 6). Pour préparer en parallèle la construction de la liste d'invariants (second sous-modèle), la nouvelle transition est communiquée à la fonction $updateMinMax$ (ligne 7). Cette fonction, qui sera détaillée à la section 3.2.3.4, met à jour les informations nécessaires au calcul des invariants qui font référence au type de l'événement qui vient d'être exécuté. La transition qui est nécessairement nouvelle est insérée dans l'ensemble des transitions qui décrivent le treillis (ligne 8). L'état atteint peut être un état déjà atteint durant la construction du treillis. Dans le cas contraire, le nouvel état est ajouté à l'ensemble des états (ligne 9) et la fonction permettant de construire le treillis est appelé récursivement à partir de ce nouvel état.

Algorithm 2 algorithme canEvolve

```
1: function CANEVOLVE(state, event)
2:     if event.type() == internal then return true
3:     else if event.type() == sending then return true
4:     else if event.type() == reception then
5:         if state.getqueue.contains(event) then return true
6:         else  return false
7:         end if
8:     end if
9: end function
```

L'algorithme 2 permet d'évaluer si la coupure qui est envisagée est cohérente ou pas. Plus précisément, cette fonction vérifie que l'ajout du nouvel événement (le paramètre *event*) à la coupure cohérente précédente (le paramètre *state*) donne une coupure cohérente. Selon la façon dont les relations de causalité entre événements sont observées, différentes stratégies peuvent être mises en œuvre. Par exemple, si un mécanisme d'horloge vectorielle a été utilisé durant l'exécution α, chaque événement de la trace est estampillé avec une valeur d'horloge logique. Dans ce cas, une simple comparaison du vecteur d'horloge associé à l'événement avec les n coordonnées associé à l'état permet de valider ou d'invalider la coupure. Le recours à un mécanisme d'horloge logique n'est pas une obligation. Ainsi, la solution présentée dans l'algorithme 2 fonctionne sans que les événements ne soient datés à condition que les deux hypothèses suivantes soient satisfaites. Premièrement, tous les événements de communication doivent être observés. Deuxièmement, pour tout événement correspondant à une réception de message, l'unique événement de la trace correspondant à l'émission de ce même message doit être identifiable. Sous ces hypothèses, si le nouvel événement exécuté par p_i est un événement local (ligne 2) ou un événement d'émission de message (ligne 3), la coupure analysée est nécessairement cohérente. Par contre, si l'événement correspond à une réception de message (ligne 4), la coupure ne sera cohérente que si l'événement correspondant à l'envoi de ce message a été préalablement exécuté (i.e., l'état *state* identifie ce message particulier comme étant un message en transit).

La construction du treillis est une opération dont la complexité (en temps et en espace mémoire) dépend en grande partie de la relation d'ordre entre les événements. Par rapport à l'exécution α, quatre valeurs numériques notées $\mathcal{S}$, $\mathcal{S}f$, $\mathcal{P}$ et $\mathcal{P}f$ sont définies à partir des cardinalités des logs ($|E^{\alpha}{}_{\mathrm{i}}|$) et à l'aide des opérateurs somme ($\sum$), produit ($\prod$) et factoriel (!) de la façon suivante :

$$\mathcal{S} = \sum_{i=1}^{n} |E^{\alpha}{}_{\mathrm{i}}| \qquad \mathcal{S}f = \mathcal{S}! \qquad \mathcal{P} = \prod_{i=1}^{n}(|E^{\alpha}{}_{\mathrm{i}}| + 1) \qquad \mathcal{P}f = \prod_{i=1}^{n}(|E^{\alpha}{}_{\mathrm{i}}|!)$$

Par définition, $\mathcal{S}$ correspond à la longueur de n'importe quel chemin du treillis (et donc de n'importe quelle séquence d'événements compatible avec l'ordre partiel $\prec^{\alpha}$). Deux cas extrêmes sont envisageables : les $\mathcal{S}$ calculs locaux peuvent être tous exécutés de façon séquentielle (le treillis ne comportera alors qu'un seul chemin) ou de façon concurrentielle (si tous les événements générés par les différents processus sont concurrents, le nombre d'états est égal à $\mathcal{P}$ et le nombre de chemins est égal à $\mathcal{S}f/\mathcal{P}f$). Les exécutions séquentielles correspondent à une complexité minimale pour l'algorithme 1. Dans ce cas, la complexité temporelle est en $\mathcal{O}(\mathcal{S})$ et la complexité spatiale est en $\mathcal{O}(\mathcal{S})$. Les exécutions concurrentielles correspondent à une complexité maximale pour l'algorithme. La complexité temporelle est en $\mathcal{O}(\mathcal{P} \times n)$ et la complexité spatiale $\mathcal{O}(\mathcal{P})$. L'algorithme est

donc d'autant plus coûteux que l'exécution est parallèle (présence d'événements concurrents).

3.2.2 Inférence d'automates

L'automate intermédiaire correspondant à l'exécution α est obtenu en construisant le treillis des états globaux cohérents à partir de la trace E^{α}. En fait, les deux structures sont équivalentes. Chaque état de l'automate intermédiaire correspond à un état global cohérent du treillis. Une transition existe entre deux états de cet automate si les coupures associées dans le treillis C_x et C_y sont telles que $C_x \cup \{e\} = C_y$. La transition est étiquetée par l'événement e dont l'exécution provoque le passage de C_x à C_y.

3.2.3 Inférence d'invariants

A ce stade, les trois classes de propriétés considérées sont celles qui ont été décrites dans la section 2.3.3 (« toujours suivi de » notée $\rightarrow$, « toujours précédé par » notée $\leftarrow$ et « jamais suivi de » notée $\nrightarrow$). Par rapport à ces trois classes, nous allons décrire une méthode (dite méthode des MinMax) qui permet d'établir efficacement la liste exhaustive des invariants caractérisant une exécution α. Dans la solution proposée, un pré-calcul est réalisé en même temps que la construction du treillis. Plus précisément, lors de la présentation de l'algorithme de construction du treillis (voir section 3.2.1), il a été mentionné qu'une fonction `updateMinMax` est appelée pour chaque nouvelle transition créée (ligne 7 de l'algorithme 1). Ces appels permettent de construire pas à pas, pour chaque type d'événement a, deux ensembles d'événements qui contiennent respectivement les événements minimaux de type a et les événements maximaux de type a.

Par la suite, ces deux nouvelles notions vont être décrites formellement et l'intérêt de ces ensembles lors du calcul des invariants va être précisé en reformulant les classes de propriétés utilisées pour le calcul des invariants. Le nom que nous avons donné à cette méthode (méthode des MinMax) découle directement des noms donnés aux deux ensembles. Dans son principe général, cette méthode présente quelques similarités avec celle décrite dans[Pfa06] pour résoudre un tout autre problème. Pour comprendre comment ces ensembles sont calculés lors de la construction du treillis des coupures cohérentes, un lien avec d'autres concepts propres aux treillis va être établi. Enfin, une description des algorithmes mis en œuvre est proposée.

3.2.3.1 Evénements minimaux et événements maximaux

A chaque type d'événements a qui se produit durant l'exécution α (*i.e.*, $a \in \overline{E^{\alpha}}$) est associé un ensemble d'événements minimaux noté $E^{\alpha}_{min}(a)$ et un ensemble d'événements

maximaux noté $E^{\alpha}_{max}(a)$. Ces deux ensembles particuliers sont des sous-ensembles de $E^{\alpha}(a)$ et correspondent respectivement à l'ensemble des événements de type a qui n'ont aucun prédécesseur de type a et à l'ensemble des événements de type a qui n'ont aucun successeur de type a.

$$E^{\alpha}_{min}(a) = \{e \in E^{\alpha}(a) \mid \forall f \in E^{\alpha}(a) - \{e\}, (e \prec^{\alpha} f) \vee (e \parallel f)\}$$

$$E^{\alpha}_{max}(a) = \{e \in E^{\alpha}(a) \mid \forall f \in E^{\alpha}(a) - \{e\}, (f \prec^{\alpha} e) \vee (e \parallel f)\}$$

Puisque $E^{\alpha}(a)$ est non vide, les deux sous-ensembles ainsi définis sont également non vides et la cardinalité de chacun d'eux ne peut excéder la valeur n puisque, par construction, chacun de ces ensembles est une antichaine (ensemble d'événements concurrents deux à deux). Bien souvent, les deux ensembles se réduisent à un singleton. C'est en particulier le cas si l'ordre observé est un ordre total (ceci se produit quand l'application synchronise fortement l'exécution des processus et ne permet pas de concurrence entre les événements) ou si tout événement de type a ne peut être produit que par un seul processus (ceci se produit lorsque chaque processus exécute un code distinct ou lorsque la définition du type d'événement cible un processus particulier). Dans ces deux cas, les événements de $E^{\alpha}(a)$ sont totalement ordonnés par la relation $\prec^{\alpha}$ et de fait, il existe un seul plus petit élément et un seul plus grand élément. Inversement, pour que l'un de ces ensembles contienne au moins deux éléments, il faut qu'au moins deux processus distincts aient exécuté de façon concurrente leurs (premiers ou derniers) événements de type a.

En reformulant les définitions des trois classes d'invariants (définitions 6, 7 et 8 présentées à la section 2.3.3), il est possible de tirer profit de ces ensembles particuliers dont la taille est généralement petite et, de toute façon, bornée par la valeur n.

3.2.3.2 Nouvelles formulations des classes d'invariants

Une nouvelle formulation des définitions des trois classes d'invariants est proposée afin de pouvoir s'appuyer sur les notions d'événements minimaux et maximaux pour établir la liste d'invariants. Dans un premiers temps, nous présentons et démontrons six équivalences entre propriétés qui vont nous servir de règles de ré-écritures. Dans chacune de ces règles, il est fait référence à un sous-ensemble non-vide de E^{α} noté E sur lequel aucune hypothèse additionnelle n'est faite. En conséquence, plusieurs règles de ré-écriture pourront être appliquées successivement pour aboutir aux nouvelles définitions des classes d'invariants.

Règle de ré-écriture R1 :

$$\forall e \in E, \exists f \in E^{\alpha}(a) \text{ tel que } e \prec^{\alpha} f \equiv \forall e \in E, \exists h \in E^{\alpha}_{max}(a) \text{ tel que } e \prec^{\alpha} h$$

Si la première propriété est satisfaite alors la seconde l'est également puisque :

$$\forall f \in E^{\alpha}(a), \exists h \in E^{\alpha}_{max}(a) \, \text{tel que} \, f = h \, \text{ou} \, f \prec^{\alpha} h$$

Si la seconde propriété est satisfaite alors la première l'est également puisque :

$$E^{\alpha}_{max}(a) \subseteq E^{\alpha}(a)$$

Règle de ré-écriture R2 :

$$\forall e \in E^{\alpha}(a), \exists f \in E \text{ tel que } e \prec^{\alpha} f \equiv \forall g \in E^{\alpha}_{max}(a), \exists f \in E \text{ tel que } g \prec^{\alpha} f$$

Si la première propriété est satisfaite alors la seconde l'est également puisque :

$$E^{\alpha}_{max}(a) \subseteq E^{\alpha}(a)$$

Si la seconde propriété est satisfaite alors la première l'est également puisque :

$$\forall e \in E^{\alpha}(a), \exists g \in E^{\alpha}_{max}(a) \, \text{tel que} \, e = g \, \text{ou} \, e \prec^{\alpha} g$$

Règle de ré-écriture R3 :

$$\forall e \in E, \exists f \in E^{\alpha}(a) \text{ tel que } f \prec^{\alpha} e \equiv \forall e \in E, \exists h \in E^{\alpha}_{min}(a) \text{ tel que } h \prec^{\alpha} e$$

Si la première propriété est satisfaite alors la seconde l'est également puisque :

$$\forall f \in E^{\alpha}(a), \exists h \in E^{\alpha}_{min}(a) \, \text{tel que} \, h = f \, \text{ou} \, h \prec^{\alpha} f$$

Si la seconde propriété est satisfaite alors la première l'est également puisque :

$$E^{\alpha}_{min}(a) \subseteq E^{\alpha}(a)$$

Règle de ré-écriture R4 :

$$\forall e \in E^{\alpha}(a), \exists f \in E \text{ tel que } f \prec^{\alpha} e \equiv \forall g \in E^{\alpha}_{min}(a), \exists f \in E \text{ tel que } f \prec^{\alpha} g$$

Si la première propriété est satisfaite alors la seconde l'est également puisque :

$$E^{\alpha}_{min}(a) \subseteq E^{\alpha}(a)$$

Si la seconde propriété est satisfaite alors la première l'est également puisque :

$$\forall e \in E^{\alpha}(a), \exists g \in E^{\alpha}_{min}(a) \, \text{tel que} \, g = e \, \text{ou} \, g \prec^{\alpha} e$$

Règle de ré-écriture R5 :

$$\forall e \in E, \, \forall f \in E^{\alpha}(a), \, f \prec^{\alpha} e \\ \equiv \\ \forall e \in E, \, \forall h \in E^{\alpha}_{max}(a), \, h \prec^{\alpha} e$$

Il s'agit d'une variante de la règle R2 et sa preuve est donc similaire.

Règle de ré-écriture R6 :

$$\forall f \in E^{\alpha}(a), \forall e \in E \text{ tel que } e \prec^{\alpha} f \\ \equiv \\ \forall g \in E^{\alpha}_{min}(a), \forall e \in E \text{ tel que } e \prec^{\alpha} g$$

Il s'agit d'une variante de la règle R4 et sa preuve est donc similaire.

En nous appuyant sur ces six règles de réécriture, nous allons reconsidérer les trois définitions de classe de propriétés présentées à la fin de la section 2.3.3 (définitions 6, 7 et 8). Ces définitions font références à deux types d'événements a et b qui se produisent durant l'exécution α.

• Pour la définition 6 (classe d'invariant $\rightarrow$), la propriété formulée est :

$$\forall e \in E^{\alpha}(a), \exists f \in E^{\alpha}(b) \text{ tel que } e \prec^{\alpha} f$$

L'application de la règle R1 donne une formulation intermédiaire :

$$\forall e \in E^{\alpha}(a), \exists h \in E^{\alpha}_{max}(b) \text{ tel que } e \prec^{\alpha} h$$

Enfin, l'application de la règle R2 donne :

$$\forall g \in E^{\alpha}_{max}(a), \exists h \in E^{\alpha}_{max}(b) \text{ tel que } g \prec^{\alpha} h$$

• De même, pour la définition 7 (classe d'invariant $\leftarrow$), la propriété initiale est :

$$\forall f \in E^{\alpha}(b), \exists e \in E^{\alpha}(a) \text{ tel que } e \prec^{\alpha} f$$

L'application de la règle R3 donne la formulation intermédiaire :

$$\forall f \in E^{\alpha}(b), \exists g \in E^{\alpha}_{min}(a) \text{ tel que } g \prec^{\alpha} f$$

Enfin, l'application de la règle R4 donne :

$$\forall h \in E^{\alpha}_{min}(b), \exists g \in E^{\alpha}_{min}(a) \text{ tel que } g \prec^{\alpha} h$$

- Dans le cas de la définition 8 (classe d'invariant $\nrightarrow$), la propriété initiale est :

$$\forall e \in E^\alpha(a), \nexists f \in E^\alpha(b) \text{ tel que } (e \prec^\alpha f) \vee (e \parallel f)$$

Elle peut se réécrire :

$$\forall e \in E^\alpha(a), \forall f \in E^\alpha(b), f \prec^\alpha e$$

L'application de la règle R5 donne la formulation intermédiaire :

$$\forall e \in E^\alpha(a), \forall h \in E^\alpha_{max}(b), h \prec^\alpha e$$

Puis, l'application de la règle R6 donne :

$$\forall g \in E^\alpha_{min}(a), \forall h \in E^\alpha_{max}(b), h \prec^\alpha g$$

Les trois définitions suivantes seront donc désormais utilisées pour calculer la valeur d'un invariant.

Définition 9 $\forall (a,b) \in \overline{E^\alpha}^2$, *$a \rightarrow b$ ssi.* $\forall e \in E^\alpha_{max}(a), \exists f \in E^\alpha_{max}(b)$ *tel que* $e \prec^\alpha f$

Définition 10 $\forall (a,b) \in \overline{E^\alpha}^2$, *$a \leftarrow b$ ssi.* $\forall f \in E^\alpha_{min}(b), \exists e \in E^\alpha_{min}(a)$ *tel que* $e \prec^\alpha f$

Définition 11 $\forall (a,b) \in \overline{E^\alpha}^2$, *$a \nrightarrow b$ ssi.* $\forall e \in E^\alpha_{min}(a), \forall f \in E^\alpha_{max}(b),\ f \prec^\alpha e$

Pour chacune des trois classes d'invariants et pour chaque paire de type d'événement (a,b), il est donc possible de déterminer si la propriété associée correspond à un invariant ou pas en considérant uniquement deux ensembles comportant au plus n événements chacun : un ensemble spécifique au type a ($E^\alpha_{min}(a)$ ou $E^\alpha_{max}(a)$) et un ensemble spécifique au type b ($E^\alpha_{min}(b)$ ou $E^\alpha_{max}(b)$).

L'approche retenue va donc consister à calculer dans un premier temps les ensembles $E^\alpha_{min}(t)$ et $E^\alpha_{max}(t)$ associés à chaque type d'évènement t de $\overline{E^\alpha}$. Puis les paires de type d'événement de $\overline{E^\alpha}^2$ vont être considérées une à une pour déterminer la liste d'invariants qui caractérise l'exécution. Pour rentabiliser le coût de calcul des ensembles $E^\alpha_{min}(t)$ et $E^\alpha_{max}(t)$, d'autres classes d'invariants peuvent être définies. Nous considérerons par la suite une quatrième classe d'invariants (notée $\rightleftharpoons$) définie de la façon suivante :

Définition 12 $\forall (a,b) \in \overline{E^\alpha}^2$, *$a \rightleftharpoons b$ ssi.* $\forall e \in E^\alpha_{min}(a), \forall f \in E^\alpha_{max}(b),\ e \prec^\alpha f$

Cette nouvelle classe d'invariants n'a pas été introduite initialement et, à notre connaissance, elle n'a pas été considérée jusqu'à présent dans la littérature. Bien que peu intuitif, l'invariant $a \rightleftharpoons b$ établit une relation forte entre les premières occurrences d'événements de type a et les dernières occurrences d'événements de type b. Chaque événement du premier groupe est toujours suivi par tous les événements du second groupe et donc, inversement, chaque événement du second groupe est toujours précédé par tous les événements du premier groupe. De fait, le nom donné à cette relation est « *premier(s) suivi(s) par dernier(s)* ». Dans le cas d'une seule exécution α, l'examen de l'invariant $a \rightleftharpoons b$ présuppose qu'au moins un événement de type a et un événement de type b se sont produits puisque $(a, b) \in \overline{E^{\alpha}}^2$. Lorsque plusieurs exécutions seront considérées, il en sera de même : $a \rightleftharpoons b$ ne peut pas être un invariant dès lors qu'une exécution au moins comporte soit des événements de type a mais pas d'événement de type b soit des événements de type b mais pas d'événement de type a. Notons également que si a est un type d'événement observé durant chaque exécution, $a \rightleftharpoons a$ peut être un invariant.

L'objectif étant de capturer des caractéristiques de l'application et non pas de spécifier une application, ces invariants plus exotiques peuvent s'avérer tout aussi intéressants. Mais surtout, la prise en compte de cette classe d'invariants additionnelle est peu coûteuse puisque l'identification de ces invariants s'appuie également sur la méthode MinMax que nous avons proposée. Une fois que tous les ensembles d'événements minimaux et maximaux ont été calculés pour chaque type d'événement, l'intérêt est de les exploiter le plus possible. Les quatre classes d'invariants retenues ne sont pas les seules qui permettent d'envisager un mode de détection s'appuyant sur la méthode MinMax. Cependant, dans la suite de cette étude, nous nous limiterons à ces quatre classes.

Pour chaque classe d'invariant et pour chaque paire de type d'événement, le test effectué consiste à vérifier l'existence de dépendances causales entre k paires d'événements issues de deux des quatre ensembles d'événement minimaux et maximaux (avec $1 \leq k \leq n^2$ puisque ces ensembles sont des antichaînes). Auparavant, le calcul des ensembles utilisés dans la méthode MinMax repose lui-aussi sur l'examen de dépendances causales entre événements. La section suivante indique comment la prise en compte des dépendances causales peut se faire en raisonnant sur les coordonnées des états du treillis plutôt que sur des horloges logiques associées aux événements.

3.2.3.3 Coordonnées des états du treillis et relation de dépendance causale

La méthode *MinMax* nécessite de tester l'existence de dépendances causales entre des occurrences d'événements. Ceci doit être fait lors de la construction des ensembles d'événements minimaux et maximaux associées à chaque type d'événement : ce calcul est réalisé en même temps que la construction du treillis. Ceci est aussi nécessaire lors

de la construction de la liste des invariants : ce calcul est réalisé après la construction du treillis.

Si une estampille est associée à chaque événement et permet de capturer les liens de causalité, il suffit d'utiliser cette information. Nous avons vu au chapitre 2 qu'une telle estampille peut être obtenue via un mécanisme de datation qui repose soit sur l'existence d'une horloge globale commune à tous les processus ou sur la mise en œuvre d'une horloge logique (horloges vectorielles [M+88] ou encore les horloges matricielles [FM82]). Par exemple, si $V(e)$ est l'horloge vectorielle associée à un événement e, de simples comparaisons entre des vecteurs de taille n permettent de détecter si des dépendances causales existent ou pas entre deux événements :

$$\forall e \in E_\alpha, \forall f \in E_\alpha - \{e\}, \quad e \prec^\alpha f \Longleftrightarrow V(e) < V(f)$$

Nous avons vu que la construction du treillis peut se faire sans disposer obligatoirement d'un mécanisme d'estampillage (voir la section 3.2.1). La question qui se pose est donc la suivante : peut-on également réaliser les tests de dépendance causale qui sont nécessaires dans la méthode *MinMax* sans utiliser d'horloge pour estampiller les événements. La solution que nous avons mise en œuvre repose sur l'utilisation des coordonnées associées à chaque état du treillis des coupures cohérentes. Nous reprenons ici la notation $D(s)$ introduite dans la section 3.2.1 pour désigner le vecteur de taille n correspondant aux coordonnées d'un état s du treillis.

D'une manière générale, il n'existe pas de lien directe entre un état (*i.e.* une coupure cohérente) et un événement particulier de l'exécution. Ceci n'est cependant pas le cas pour deux types particuliers d'états du treillis que nous allons maintenant introduire, à savoir les états « join irreducible » et les états « meet irreducible » [War38, Kom43]. Pour définir ces états particuliers, nous devons d'abord introduire la notion de couverture dans un treillis des coupures cohérentes $(\mathcal{C}^\alpha, \subseteq)$ associé à une exécution α d'une application répartie.

Définition 13 *$\forall C_x, C_y, C_z \in \mathcal{C}^\alpha$ tels que $C_x \neq C_y$, on dit que C_x est couvert par C_y, ou que C_y couvre C_x, si $C_x \subseteq C_y$, et $C_x \subseteq C_z \subseteq C_y$ implique $C_x = C_z$ ou $C_z = C_y$.*

Lorsque l'état C_x est couvert par l'état C_y, l'état C_y peut alors être noté sous la forme $C_x \cup \{evt(C_x, C_y)\}$ où $evt(C_x, C_y)$ est l'unique événement de l'application qui appartient à la coupure C_y mais qui n'appartient pas à la coupure C_x.

Un état $C_x \in \mathcal{C}^\alpha$ est dit « join irreducible » s'il couvre exactement un autre état. Informellement, cela signifie que l'état n'a qu'une seule transition entrante dans le treillis. Sans connaître l'identité de l'unique état couvert par C_x, nous utilisons la notation

$evt(-, C_x)$ pour désigner l'événement particulier ainsi associé à C_x. De même, un état $C_x \in \mathcal{C}^\alpha$ est dit « meet irreducible » s'il est couvert par un seul état. Informellement, cela signifie que l'état n'a qu'une seule transition sortante dans le treillis. De la même manière, sans connaître l'identité de l'unique état qui couvre C_x, la notation $evt(C_x, -)$ désignera l'événement particulier ainsi associé à C_x. L'ensemble de tous les états « join irreducible » est noté JI^α et l'ensemble de tous les états « meet irreducible » est noté MI^α. Dans un treillis fini, le nombre d'états « meet irreducible » est le même que le nombre d'états « join irreducible » [Dil90]. A chaque événement de l'application peut être associé un état de JI^α et un état de MI^α qui sont nécessairement distincts. Si les deux ensembles JI^α et MI^α ont la même cardinalité, ils ne sont par contre pas nécessairement disjoints et leur union n'inclut pas tous les états du treillis.

Puisqu'un événement unique peut être associé à chaque état des ensembles JI^α et MI^α, il est possible d'établir des liens entre les coordonnées de ces états dans le treillis et les valeurs des horloges logiques des événements associés. En particulier :

$$\forall C_x \in JI^\alpha, D(C_x) = V(evt(-, C_x))$$

Autrement dit, les coordonnées d'un état « join irreducible » correspondent exactement à la valeur de l'horloge vectorielle de l'événement associé. Par conséquent :

$$\forall C_x \in JI^\alpha, \forall C_y \in JI^\alpha - \{C_x\}, \quad evt(-, C_x) \prec^\alpha evt(-, C_y) \Longleftrightarrow D(C_x) < D(C_y)$$

En ce qui concerne les états de MI^α, leur coordonnées sont également encadrées par les propriétés suivantes :

$$\forall C_x \in MI^\alpha, \forall e_i^y \in E_i^\alpha \text{ tel que } evt(C_x, -) = e_i^y \text{ ou } evt(C_x, -) \prec^\alpha e_i^y, D(C_x)[i] < V(e_i^y)[i]$$

$$\forall C_x \in MI^\alpha, \forall e \in E^\alpha \text{ tel que } e \prec^\alpha evt(C_x, -), \quad D(C_x) > V(e)$$

$$\forall C_x \in MI^\alpha, \forall e \in E^\alpha \text{ tel que } e \parallel^\alpha evt(C_x, -), \quad D(C_x) \geq V(e)$$

Par conséquent :

$$\forall C_x \in MI^\alpha, \forall C_y \in MI^\alpha - \{C_x\}, \quad evt(C_x, -) \prec^\alpha evt(C_y, -) \Longleftrightarrow D(C_x) < D(C_y)$$

Les coordonnées des états de JI^α et MI^α permettent donc de capturer les dépendances causales entre les événements qui leur sont associés.

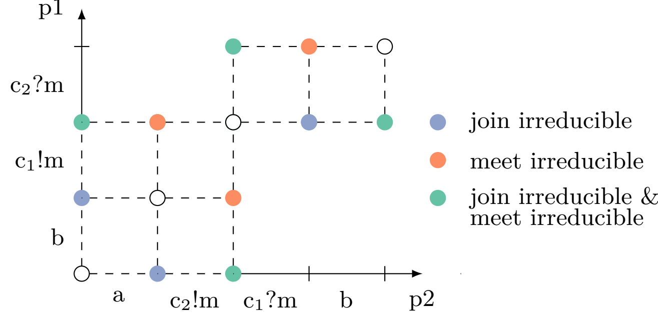

FIGURE 3.2 – Etats Meet & Join irreducible correspondant à l'exécution α

Considérons l'exemple exposé dans la section 2.2, les états « join irreducible » et « meet irreducible » pour l'exécution α sont représentés dans la figure 3.2. Les ensembles d'états JI^α et MI^α sont chacun composés de sept états. A noter que les coordonnées (x, y) d'un état font référence au processus p_1 (x sur l'axe vertical) et au processus p_2 (y sur l'axe horizontale). Si on considère l'état qui a pour coordonnées $(0, 1)$, il couvre seulement l'état (0,0). L'état de coordonnées $(0, 1)$ est donc un état « join irreducible » qui est associé à l'événement a. Les coordonnées de cet état correspondent à l'horloge vectorielle de l'événement a. L'événement a est également associé à un état « meet irreducible » qui a pour coordonnées $(2, 0)$ et qui est couvert uniquement par l'état $(2, 1)$. Certains états, comme par exemple l'état de coordonnées $(3, 2)$ appartiennent à la fois à JI^α et MI^α. En tant qu'élément de JI^α, cet état est rattaché à l'événement $c2?m$ de p_1 et, en tant qu'élément de MI^α, il est associé à l'événement $c1?m$ de p_2. Chaque état « join irreducible » (resp. « meet irreducible ») du treillis peut être comparé avec les autres « join irreducible » (resp. « meet irreducible »). Par exemple, il est possible de comparer les coordonnées des états « join irreducible » associés à l'événement a $(0, 1)$ et à l'événement $c2?m$ $(3, 2)$ pour en déduire qu'ils sont causalement dépendants.

3.2.3.4 Calcul des ensembles minimaux et maximaux

Rappelons que l'objectif est de construire, pour chaque type d'événement $a \in \overline{E^\alpha}$, un ensemble $E^\alpha_{max}(a)$ et un ensemble $E^\alpha_{min}(a)$ qui respectent les définitions vues au début de la section 3.2.3.1 :

$$E^\alpha_{max}(a) = \{e \in E^\alpha(a) \mid \forall f \in E^\alpha(a) - \{e\},\, f \prec^\alpha e \text{ ou } e \parallel^\alpha f\}$$
$$E^\alpha_{min}(a) = \{e \in E^\alpha(a) \mid \forall f \in E^\alpha(a) - \{e\},\, e \prec^\alpha f \text{ ou } e \parallel^\alpha f\}$$

Pour cela, la méthode proposée consiste à calculer deux ensembles $MaxEvtList^\alpha(a)$ et $MinEvtList^\alpha(a)$ qui sont des sur-approximations des deux ensembles que nous souhaitons calculer.

$$MaxEvtList^\alpha(a) = \{e_i^x \in E^\alpha(a) \mid \forall f \in E^\alpha(a) \cap E_i^\alpha, f \prec^\alpha e_i^x \text{ ou } e_i^x = f\}$$
$$MinEvtList^\alpha(a) = \{e_i^x \in E^\alpha(a) \mid \forall f \in E^\alpha(a) \cap E_i^\alpha, e_i^x \prec^\alpha f \text{ ou } e_i^x = f\}$$

Si au moins un événement de type a se produit durant l'exécution du processus p_i alors l'ensemble $MaxEvtList^\alpha(a)$ (respectivement $MinEvtList^\alpha(a)$) contient la dernière (respectivement la première) occurrence d'événement de type a observée sur le processus p_i. Par définition, les ensembles $MaxEvtList^\alpha(a)$ et $MinEvtList^\alpha(a)$ contiennent au plus un événement produit par chaque processus. Ces ensembles ne correspondent pas nécessairement à des antichaînes car deux événements de type a produits sur deux processus différents peuvent figurer dans un de ces ensembles alors qu'ils sont causalement dépendants.

L'obtention de ces ensembles d'événements va elle aussi reposer sur le calcul (tout au long de la construction du treillis) de deux ensembles d'états $MaxStList^\alpha(a)$ et $MinStList^\alpha(a)$ qui sont chacun des sous ensembles de MI^α.

$$MaxStList^\alpha(a) = \{C_x \in MI^\alpha \mid evt(C_x, -) \in MaxEvtList^\alpha(a)\}$$
$$MinStList^\alpha(a) = \{C_x \in MI^\alpha \mid evt(C_x, -) \in MinEvtList^\alpha(a)\}$$

Lors de la construction du treillis (voir la section 3.2.1), à chaque découverte d'une nouvelle transition dans le treillis, la fonction *updateMinMax* est appelée (ligne 7 de l'algorithme 1). L'algorithme 3 décrit le code de cette fonction. Dans la suite de cette explication, nous supposons que la transition qui est fournie en paramètre lors de chaque appel (ligne 1) permet de transiter dans le treillis d'un état s_1 à un état s_2 et que ce changement d'état correspond à l'exécution par un processus p_j d'un événement e de type $\overline{e}$. A une transition correspond un événement et à chaque événement correspond une ou plusieurs transitions. L'algorithme utilisé ne fait aucune hypothèse sur l'ordre dans lequel les transitions sont traitées : seul compte le fait que toutes les transitions du treillis vont être analysées.

Algorithm 3 algorithme updateMinMax

```
1: function UPDATEMINMAX(transition)
2:     minmax ← minmaxmap.get(transition.getevent())
3:     if minmax == null then
4:         minmax ← MinMax(transition)
5:         minmaxmap.put(transition.getevent(), minmax)
6:     else
7:         minmax.updateMax(transition)
8:         minmax.updateMin(transition)
9:     end if
10: end function
```

A la ligne 2, les informations relatives à la transition sont utilisées pour accéder à une structure de donnée (nommée *minmax*) qui pointe sur les deux listes associées au type d'événement $\overline{e}$, à savoir $MaxStList^{\alpha}(\overline{e})$ et $MinStList^{\alpha}(\overline{e})$. Deux cas sont possibles. Soit le type d'événement $\overline{e}$ est rencontré pour la première fois. Dans ce cas, la structure *minmax* correspondante ne pointe sur aucune liste ($minmax == null$). Soit le type d'événement $\overline{e}$ a déjà été vu lors du traitement de précédente(s) transition(s) liée(s) à l'événement e ou lors du traitement d'un autre événement f tel que $\overline{f} = \overline{e}$.

Dans le premier cas (lignes 3 à 5 de l'algorithme 3), l'état s_1 va être utilisé pour initialiser les listes $MaxStList^{\alpha}(\overline{e})$ et $MinStList^{\alpha}(\overline{e})$ avec le même élément. Sans entrer dans les détails, l'élément créé contient notamment les coordonnées $D(s_1)$ et l'identité du processus p_j ayant produit l'événement e. L'événement e étant le tout premier événement de type $\overline{e}$ pris en compte, l'état d'origine de la transition est considéré à la fois comme l'état minimum et l'état maximum associé au type d'événement de type $\overline{e}$. Bien entendu, à ce stade, rien ne permet d'affirmer que l'élément ajouté est un état « meet irreducible ».

Dans le second cas (lignes 6 à 8 de l'algorithme 3), les listes ne sont pas vides et doivent peut-être être modifiées. La mise à jour de la liste $MaxStList^{\alpha}(\overline{e})$ qui est faite à la ligne 7 via un appel à la fonction *updateMax* et la mise à jour de la liste $MinStList^{\alpha}(\overline{e})$ qui est faite à la ligne 8 via un appel à la fonction *updateMin* sont indépendantes. Puisque le principe est le même, seul le code correspondant à la première fonction de mise à jour nommée *updateMax* est décrite dans l'algorithme 4.

Algorithm 4 algorithme updateMax

```
 1: function UPDATEMAX(transition)
 2:     comparable ← false
 3:     for i in this.getmaxlist() do
 4:         transtemp ← this.getmaxlist().get(i)
 5:         if sameProcess(transtemp, transition)& comparable == false then
 6:             if transtemp.getPrevState() <= transition.getPrevState() then
 7:                 updateMaxList(i, transition, updated)
 8:                 comparable ← true
 9:             else if transtemp.getPrevState() >= transition.getPrevState() then
10:                 comparable ← true
11:             end if
12:         end if
13:     end for
14:     if comparable == false then
15:         this.getmaxlist().add(transition)
16:     end if
17: end function
```

La fonction $updateMax$ prend en paramètre la transition nouvellement créée. Dans une boucle itérative (de la ligne 3 à la ligne 13), tous les éléments déjà présents dans la liste $MaxStList^{\alpha}(\overline{e})$ sont successivement considérés. Chaque élément présent correspond à une transition qui a été analysée auparavant et qui était associée à un événement f de même type ($\overline{f} = \overline{e}$) produit par un processus p_i (ligne 4). Le test principal (ligne 5) consiste à vérifier si p_i est aussi l'identité du processus qui a produit l'événement e associée à la nouvelle transition en cours d'analyse. Pour rappel, dans notre explication, ce processus particulier est supposé être p_j. Si ce n'est pas le cas ($p_i \neq p_j$), aucune mise à jour n'est effectuée dans le corps de la boucle. Au final, deux cas mutuellement exclusifs sont à considérer. Soit aucun des éléments déjà existant dans $MaxStList^{\alpha}(\overline{e})$ ne concerne p_j. Soit un élément (et nécessairement un seul) existe déjà et concerne également p_j. Dans le premier cas, la variable booléenne *comparable* reste à faux durant tout le calcul et un nouvel élément correspondant à la nouvelle transition est créé et ajouté à la ligne 15. Dans le second cas, les coordonnées de l'état associé à l'élément existant sont comparés avec les coordonnées de $D(s_1)$. Puisque p_i et p_j correspondent au même processus, l'une des trois relations suivantes est nécessairement vrai : $e = f$, $e \prec^{\alpha} f$ ou $f \prec^{\alpha} e$. Même si, à ce stade, les états ne sont pas forcement des éléments de JI^{α} ou de MI^{α}, la comparaison de leurs vecteurs de coordonnées (et notamment l'entrée i de ces deux vecteurs) permet de déterminer précisément la relation qui est vraie parmi les trois possibilités. En conséquence, il est possible sur la base des coordonnées des deux

états de décider s'il faut conserver l'ancien élément et ignorer le nouveau (lignes 9 et 10) ou, au contraire, le remplacer par le nouvel élément (lignes 6 à 8).

Pour chaque transition d'un état s_1 vers un état s_2, nous nous focalisons sur l'état s_1 qui peut être un potentiel « meet irreducible ». Cependant, tant que toutes les transitions n'ont pas été vues, les éléments des ensembles construits ne sont pas nécessairement des états « meet irreducible ». Autrement dit, tant que le treillis n'est pas complet, les définitions qui ont été données pour les ensembles $MaxStList^{\alpha}(\overline{e})$ et $MinStList^{\alpha}(\overline{e})$ ne sont pas respectées car d'une part des états qui doivent y figurer peuvent être temporairement manquants et d'autre part des états qui n'ont pas à y figurer peuvent être temporairement présents. Mais une fois le treillis entièrement construit, chaque type d'événement qui se produit durant l'exécution est associé à deux ensembles d'états corrects dont les éléments sont tous des états « meet irreducible ». Dès lors que les ensembles $MaxStList^{\alpha}(\overline{e})$ et $MinStList^{\alpha}(\overline{e})$ ne contiennent que des états de MI^{α}, un événement unique $(evt(C_x, -))$ est associé à chaque état C_x : Il est donc possible de passer de l'ensemble $MaxStList^{\alpha}(\overline{e})$ (respectivement $MinStList^{\alpha}(\overline{e})$) à l'ensemble $MaxEvtList^{\alpha}(\overline{e})$ (respectivement $MinEvtList^{\alpha}(\overline{e})$). Puis, puisque les dépendances causales entre événements sont détectables en comparant les coordonnées des états « meet irreducible » qui leur correspondent, une phase ultime permet d'identifier au sein de ces deux ensembles d'au plus n événements les deux sous-ensembles d'événements maximaux et minimaux : $E^{\alpha}_{max}(\overline{e})$ et $E^{\alpha}_{min}(\overline{e})$.

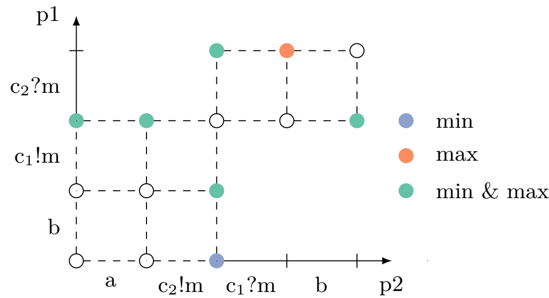

FIGURE 3.3 – Représentation des transitions Min et Max

Si on considère à nouveau l'exemple détaillé dans la section 2.2 qui comporte sept événements, le calcul des ensembles $MaxStList^{\alpha}(\overline{e})$ et $MinStList^{\alpha}(\overline{e})$ donne en résultat douze ensembles puisque les événements sont de six types différents. Les deux événements de type b (à savoir e^1_1 et e^4_2) sont causalement dépendants. En conséquence, dans cet exemple particulier, les ensembles calculés sont tous des singletons. Dans la figure 3.3 sont représentés les six états (appelés max) correspondants à des ensembles $MaxStList^{\alpha}(\overline{e})$ et

les six états (appelés min) correspondants à des ensembles $MinStList^{\alpha}(\overline{e})$. Puisque cinq types d'événements ne sont associés qu'à un seul et unique événement, il y a cinq états qui sont à la fois inclus dans un ensemble $MaxStList^{\alpha}(\overline{e})$ et l'ensemble $MinStList^{\alpha}(\overline{e})$ associé. En ce qui concerne le type b, l'état de coordonné $(0, 2)$ et l'état de coordonné $(3, 3)$ sont contenus respectivement dans $MinStList^{\alpha}(b)$ et $MaxStList^{\alpha}(b)$. Visuellement, on peut constater que les sept états identifiés sont bien des états « meet irreducible » du treillis : chaque état n'a qu'une seule transition sortante.

Analyse de la complexité : Lors de l'exécution α qui implique n processus, nous supposons que le treillis complet comprend t transitions et que le nombre de types d'événement différents qui se produisent est noté $\mathcal{T} = |\overline{E^{\alpha}}|$. Les calculs effectués après la construction du treillis nécessitent de vérifier dans $2\mathcal{T}$ ensembles d'au plus n événements que les évènements sont concurrents deux à deux. La complexité de cette phase de calcul est donc en $\mathcal{O}(n^2 \times \mathcal{T})$. La phase préliminaire qui consiste à mettre à jour les $2\mathcal{T}$ ensembles d'états potentiellement « meet irreducible » nécessite d'effectuer un calcul à chaque nouvelle transition. Ce calcul débute par l'identification des deux ensembles associés au type d'événement qui caractérise la transition. Dans notre mise en œuvre, cette recherche a un coût en $\mathcal{O}(log(\mathcal{T}))$. La mise à jour d'un ensemble nécessite d'examiner jusqu'à n éléments. En conséquence, cette première phase du calcul des états associés aux événements minimaux et maximaux a une complexité temporelle en $\mathcal{O}(t \times log(\mathcal{T}) \times n)$. Les calculs ont une complexité spatiale en $\mathcal{O}(\mathcal{T} \times n)$.

3.2.3.5 Inférence des invariants

Une fois que les ensembles d'événements minimaux et maximaux ont été établis pour chaque type d'événement, une deuxième étape consiste à utiliser ces ensembles pour déterminer quelles sont les propriétés qui ont été satisfaites sur toute la durée de l'exécution et qui seront donc considérées comme des probables-invariants.

Pour rappel, nous avons retenu quatre classes de propriétés (dont les définitions 9, 10, 11 et 12 apparaissent à la section 3.2.3.2). Supposons que a et b sont deux types d'événement observés durant l'exécution α : $(a, b) \in \overline{E^{\alpha}}^2$. Les tests peuvent porter sur les horloges logiques des événements des événements minimaux et maximaux.

- $a \rightarrow b$ ssi. $\forall e \in E^{\alpha}_{max}(a), \exists f \in E^{\alpha}_{max}(b)$ tel que $V(e) < V(f)$
- $a \leftarrow b$ ssi. $\forall f \in E^{\alpha}_{min}(b), \exists e \in E^{\alpha}_{min}(a)$ tel que $V(e) < V(f)$
- $a \nrightarrow b$ ssi. $\forall e \in E^{\alpha}_{min}(a), \forall f \in E^{\alpha}_{max}(b)$, $V(f) < V(e)$
- $a \rightleftharpoons b$ ssi. $\forall e \in E^{\alpha}_{min}(a), \forall f \in E^{\alpha}_{max}(b)$, $V(e) < V(f)$

Puisque tous les événements minimaux et maximaux sont associés à un état de MI^{α} dont les coordonnées sont connues, il est possible de raisonner sur les coordonnées de ces états. Nous utilisons à nouveau la notation $evt(C_z, -)$ pour désigner l'événement associé à l'unique transition qui est issue d'un état C_z de MI^{α}.

— $a \rightarrow b$ ssi. $\forall\, evt(C_x, -) \in E^{\alpha}_{max}(a), \exists\, evt(C_y, -) \in E^{\alpha}_{max}(b)$ tel que $D(C_x) < D(C_y)$

— $a \leftarrow b$ ssi. $\forall\, evt(C_y, -) \in E^{\alpha}_{min}(b), \exists\, evt(C_x, -) \in E^{\alpha}_{min}(a)$ tel que $D(C_x) < D(C_y)$

— $a \nrightarrow b$ ssi. $\forall\, evt(C_x, -) \in E^{\alpha}_{min}(a), \forall\, evt(C_y, -) \in E^{\alpha}_{max}(b)$, $D(C_y) < D(C_x)$

— $a \rightleftharpoons b$ ssi. $\forall\, evt(C_x, -) \in E^{\alpha}_{min}(a), \forall\, evt(C_y, -) \in E^{\alpha}_{max}(b)$, $D(C_x) < D(C_y)$

L'algorithme 5 détaille la méthode de calcul des invariants. Deux boucles imbriquées permettent d'énumérer tous les couples (a, b) de types d'évènements qui se sont produits durant l'exécution α. Pour chaque couple, les quatre classes d'invariants sont examinées les unes après les autres (lignes 6 à 8 pour l'invariant $a \rightarrow b$, lignes 9 à 11 pour l'invariant $a \leftarrow b$, lignes 12 à 14 pour l'invariant $a \nrightarrow b$ et lignes 15 à 17 pour l'invariant $a \rightleftharpoons b$). Pour chaque invariant, une fonction spécifique est appelée pour vérifier à partir des ensembles minimaux et/ou maximaux, si le test que nous avons détaillé précédemment est satisfait ou pas. Si la condition testée est vérifiée, le couple (a, b) est ajouté à la liste des couples qui satisfont la classe d'invariant testée.

A noter que les tests sont effectués de façon indépendante et systématique. Dans certains cas, il est cependant possible de tirer profit de la connaissance du résultat d'un test pour en déduire automatiquement les résultats d'autres tests. Par exemple, si l'invariant $a \nrightarrow b$ est vérifié, il est possible de déduire que les invariants $b \leftarrow a$ et $b \rightarrow a$ le sont aussi. En effet, si $a \nrightarrow b$ est un invariant alors tous les événements de $E^{\alpha}_{max}(b)$ précèdent causalement tous les événements de $E^{\alpha}_{min}(a)$ (d'après la propriété 11). Il est donc possible de conclure que, a fortiori, tous les éléments de $E^{\alpha}_{max}(b)$ précèdent causalement tous les éléments de $E^{\alpha}_{max}(a)$ et, par conséquent, que $b \rightarrow a$. De même, il est possible de conclure que, a fortiori, tous les éléments de $E^{\alpha}_{min}(b)$ précèdent causalement tous les éléments de $E^{\alpha}_{min}(a)$ et, par conséquent, que $b \leftarrow a$. Nous reviendrons sur ces optimisations lorsque nous aborderons la notion de redondance au sein des listes d'invariants. D'autres optimisations plus triviales concernent l'évaluation des prédicats en lien avec des couples de type d'événement identique tel que (a, a). Nous avons déjà mentionné que les propriétés $a \rightarrow a$ et $a \leftarrow a$ sont nécessairement fausses. Par ailleurs, des propriétés sont mutuellement exclusives : par exemple si $a \rightarrow b$ est satisfaite alors la propriété $b \rightarrow a$ est nécessairement fausse. En conséquence, la complexité temporelle de cette phase d'inférence des invariants est en $\mathcal{O}(\mathcal{T}^2 \times n)$. Cependant, pour certaines classes de propriété, la complexité peut être réduite et être seulement en $\mathcal{O}(((\mathcal{T} \times (\mathcal{T} - 1))/2) \times n)$.

Algorithm 5 algorithme de calcul des invariants

```
1:  function INVINFERENCE(minmaxmap)
2:      for a in typelist do
3:          mmE1 ← minmaxmap.get(a)
4:          for b in typelist do
5:              mmE2 ← minmaxmap.get(b)
6:              if alwaysFollowedBy(mmE1.getMax(), mmE2.getMax()) then
7:                  afblist.add(a, b)
8:              end if
9:              if alwaysPrecededBy(mmE1.getMin(), mmE2.getMin()) then
10:                 apbblist.add(a, b)
11:             end if
12:             if neverFollowedBy(mmE1.getMin(), mmE2.getMax()) then
13:                 nfblist.add(a, b)
14:             end if
15:             if firstFBLast(mmE1.getMin(), mmE2.getMax()) then
16:                 ffbllist.add(a, b)
17:             end if
18:         end for
19:     end for
20: end function
```

3.3 Fusion de modèles intermédiaires et généralisation

La première étape de la phase de construction visait à inférer des modèles intermédiaires à partir d'un ensemble fini de traces d'exécutions. Un modèle dual (incluant un automate et une liste d'invariants) a été construit pour chaque exécution. Cette collection de modèles intermédiaires est utilisée ensuite pour construire un modèle unique du comportement normal de l'application. Cette construction du modèle global est réalisée lors de la seconde étape de la phase de construction à l'aide du module *Model Merger & Generalizer* (voir figure 3.1). Le modèle obtenu sera dual lui-aussi et donc composé d'un automate global généralisé et d'une liste d'invariants globaux. Les différentes étapes pour générer un modèle global à partir de modèles intermédiaires sont décrites dans les sections suivantes.

3.3.1 Fusion d'automates et généralisation

Nous disposons de plusieurs automates intermédiaires obtenus à partir des différentes traces d'exécutions. L'objectif est d'obtenir un automate unique qui accepte tous les comportements de l'application qui sont reconnus par au moins un des automates

intermédiaires. Cet objectif est facilement atteint en fusionnant les états initiaux des différents automates.

Après cette fusion des états initiaux, l'automate obtenu accepte toutes les traces utilisées pour construire le modèle lors de la phase d'apprentissage et refuse tout comportement qui n'a pas été observé au cours de ces exécutions. Cependant, il peut être difficile d'exhiber l'ensemble des comportements normaux possibles lors de la phase d'apprentissage. Une phase de généralisation est alors souvent utile pour introduire de nouveaux comportements, raisonnablement proches des comportements appris. La généralisation dans notre approche est réalisée à l'aide de l'algorithme k-tail (présenté à la section 2.4.1). Cet algorithme est connu pour produire des modèles assez précis pour des applications simples mais peu précis pour des application complexes [LK06]. Dans un premier temps, nous considérons deux implémentations possibles de l'algorithme k-tail : la première implémentation (exacte) correspond à la proposition de k-tail présente dans l'article [BF72] ; la seconde implémentation de k-tail (inclusive) reprend le fonctionnement général de k-tail mais utilise une relation d'inclusion pour les k futures transitions [LMP08]. Notre implémentation de k-tail se décompose en deux étapes détaillées ci-dessous.

Algorithm 6 identification des états à fusionner

```
function identKSimilarStates(statelist, k, mode)
    for i in statelist.size() do
        for j in statelist.size() do
            if iskSimilar(statelist.get(i), statelist.get(j), k, mode) then
                couplelist.add(statelist.get(i), statelist.get(j))
                statelist.remove(j)
            end if
        end for
    end for
end function
```

L'algorithme 6 constitue la première étape de l'algorithme k-tail. Il vise à comparer tous les états entre eux à l'aide de leurs k futurs pour déterminer les états qui seront à fusionner lors de la seconde étape. Le k futur d'un état correspond à tous les enchaînements d'au plus k tirages de transitions qui sont possibles à partir de cet état : chaque transition étant associée à un type d'événement, ce sont donc des suites d'au plus k types d'événements qui constituent un k futur. L'algorithme prend en paramètre l'ensemble des états de l'automate (représenté par la variable *statelist*), le paramètre k de k-tail et le mode de comparaison (exact ou inclusif). Dans cet algorithme, toutes les paires (non ordonnées) d'états sont analysées à l'aide de la méthode $isKsimilar$. Pour simplifier la présentation, nous avons ici considéré une double boucle imbriquée mais en

réalité, chaque pair d'états (i, j) n'est évaluée qu'une seule fois. La méthode compare les k futurs de deux états selon deux modes. Si le mode choisi est "exact", les k futurs des deux états doivent être exactement les mêmes. Si le mode de fusion est "inclusif", le k futur d'un des deux états doit être inclus dans le k futur de l'autre état sans qu'ils soient nécessairement égaux. Le premier mode est donc plus contraignant que le second. Dans la première étape, toutes les paires d'états k équivalents sont identifiées sans que les fusions ne soient réalisées.

Algorithm 7 fusion des états similaires

```
function mergeKSimilarStates(k)
    for i in couplelist.size() do
        mergekSimilar(couplelist.get(i), k)
    end for
end function
```

L'algorithme 7 correspond à la deuxième étape de l'algorithme. Ici, tous les états préalablement sélectionnés comme k équivalents sont fusionnés. Cette fusion est réalisée à l'aide de la fonction $mergeKSimilar$. Les couples d'états identifiés comme k équivalents voient leurs états et transitions similaires fusionnés à profondeur k. De fait, lorsque la valeur de k est supérieure à 1, la prise en compte de deux états équivalents se traduit potentiellement par la fusion de plus de deux états.

Pour illustrer le comportement de l'algorithme, nous considérons que la phase d'apprentissage ne tient compte que d'une seule trace : celle correspondant à l'exécution α. L'automate qui va être généralisé est donc l'automate intermédiaire représenté dans la figure 2.5.

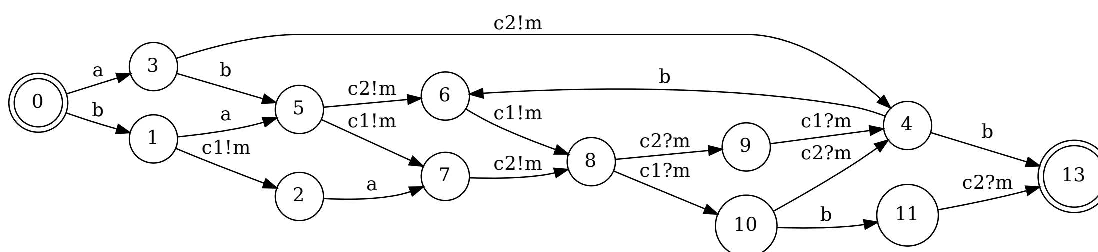

FIGURE 3.4 – Généralisation en mode exact de l'automate de l'exécution α avec $k = 1$

La figure 3.4 présente une généralisation de cet automate en adoptant un k égal à 1 et un mode exact. L'algorithme 6 identifie les états "4" et "12" comme étant k-similaires. Ces deux états présentent exactement les mêmes 1-future transitions (voir figure 2.5).

L'algorithme 7 va fusionner les deux états en modifiant les transitions entrantes et sortantes du deuxième état et en fusionnant les transitions sortantes correspondant aux mêmes événements.

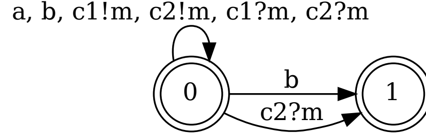

FIGURE 3.5 – Généralisation en mode inclusif de l'automate de l'exécution α avec $k = 1$

La figure 3.5 présente quant à elle une généralisation du même automate en adoptant également un k égal à 1 mais en utilisant cette fois le mode inclusif. A l'évidence, les fusions sont beaucoup plus nombreuses. Sur ce simple exemple, nous voyons les limites de ce deuxième mode : l'automate obtenu permet de vérifier que les évènements observés sont d'un type connu (*i.e.* leurs types doivent faire partie de l'alphabet associé à l'automate généralisé) mais les contraintes portant sur l'enchaînement des types d'événement sont très faibles. Dans l'exemple, la seule contrainte résiduelle est que l'enchaînement doit se terminer soit par un événement de type b, soit par un événement de type $c2?m$. A moins de vouloir fortement diminuer le taux de faux positif, l'intérêt du mode inclusif nous est apparu assez limité au cours des différents tests réalisés. Nous ferons uniquement référence au mode exact dans la suite de ce manuscrit.

Notre implémentation de l'algorithme k-tail a une complexité temporelle en $\mathcal{O}((\mathcal{E} \times (\mathcal{E} - 1))/2)$ où $\mathcal{E}$ correspond au nombre total d'états de l'automate avant généralisation. La complexité de la seconde phase dépend uniquement du nombre de paires d'états identifiées comme étant k-équivalent. Dans le pire cas, les $\mathcal{E} \times (\mathcal{E} - 1))/2$ paires d'états sont toutes concernées.

3.3.2 Fusion d'invariants

La fusion d'invariants s'opère en utilisant toutes les listes intermédiaires d'invariants qui ont été précédemment calculées de façon séparée. Pour rappel, une liste intermédiaire correspond à une exécution particulière étudiée durant la phase d'apprentissage. L'objectif de la fusion est d'obtenir une unique liste d'invariants qui représente l'ensemble des comportements normaux appris. Cette fusion est réalisée en deux étapes : une première étape consiste à fusionner les listes d'invariants intermédiaires qui concernent différentes

exécutions pour obtenir une liste d'invariants satisfaits par toutes les exécutions (ou plus précisément, satisfaits par au moins une exécution et violé lors d'aucune autre) ; une seconde étape vise à simplifier cette liste d'invariants en enlevant notamment les invariants redondants.

Première étape :

L'étape de fusion des listes d'invariants intermédiaires est réalisée en fusionnant les listes deux à deux. Pour décrire cette étape, nous considérons deux exécutions E_α et E_β et les listes d'invariants I_α et I_β correspondant à ces deux exécutions. $I_{\alpha\cap\beta}$ correspond à la liste d'invariants obtenue suite à la fusion des deux listes.

Pour fusionner deux listes, on peut envisager plusieurs approches. La première proposition consiste à considérer qu'un invariant est vrai à l'issue de la fusion uniquement s'il est contenu dans les listes intermédiaires d'invariants des deux exécutions. Cette proposition est simple et intuitive. Par exemple, si on a : $a \rightarrow b \in I_\alpha$ et $a \rightarrow b \in I_\beta$, l'invariant $a \rightarrow b$ doit nécessairement figurer dans $I_{\alpha\cap\beta}$. Mais cette approche s'avère trop restrictive quand le comportement d'une application varie sensiblement d'une exécution à une autre.

Ainsi, des types d'événements peuvent être propres à certaines exécutions et ne pas apparaître dans d'autres : autrement dit, les ensembles $\overline{E^\alpha}$ et $\overline{E^\beta}$ ne sont pas nécessairement égaux ou inclus l'un dans l'autre. Les invariants calculés qui concernent deux types d'événements spécifiques à une exécution ne doivent pas systématiquement être considérés comme violés durant une exécution où les même types d'événements n'apparaissent pas. Par exemple, si l'invariant $a \rightarrow b$ est dans la liste d'invariants I_α mais que les types d'événements a et b n'apparaissent pas durant l'exécution β, l'invariant ne figure pas dans la liste I_β car la construction d'une liste intermédiaire ne tient compte que des couples de types d'événements observés durant l'exécution. Mais pour autant l'invariant n'est pas violé par la deuxième exécution : il n'existe aucune occurrence d'un événement de type a qui ne soit pas suivie par au moins un événement de type b.

De même, si un invariant, vérifié lors d'une exécution, implique deux types d'événement et que l'un est observé lors des deux exécutions alors que l'autre type n'est présent que dans une, l'invariant peut, suivant les cas, être ou ne pas être conservé lors de la fusion. Par exemple, si l'invariant $a \rightarrow b$ est dans la liste d'invariants I_α, et que le type d'événement a n'appartient pas à E_β alors que le type d'événement b appartient à E_β, l'invariant $a \rightarrow b$ restera vrai pour la seconde exécution. En revanche, cet invariant sera violé si le type d'événement a est présent dans E_β mais que b n'y est pas présent. Pour réaliser la fusion des listes d'invariants deux à deux, il faut donc convenir de règles spécifiques à chaque classe d'invariants. Pour les quatre classes considérées, nous utilisons les définitions suivantes :

Définition 14 $\forall\ a \in \overline{E^\alpha} \cup \overline{E^\beta},\ \forall\ b \in \overline{E^\alpha} \cup \overline{E^\beta},$ *"a → b" ∈* $I_{\alpha\cap\beta}$ *ssi :*

- $"a \rightarrow b" \in I_\alpha \wedge "a \rightarrow b" \in I_\beta$
- $"a \rightarrow b" \in I_\alpha \wedge a \notin \overline{E^\beta}$

Définition 15 $\forall\ a \in \overline{E^\alpha} \cup \overline{E^\beta},\ \forall\ b \in \overline{E^\alpha} \cup \overline{E^\beta},$ *"a ← b" ∈* $I_{\alpha\cap\beta}$ *si :*

- $"a \leftarrow b" \in I_\alpha \wedge "a \leftarrow b" \in I_\beta$
- $"a \leftarrow b" \in I_\alpha \wedge b \notin \overline{E^\beta}$

Définition 16 $\forall\ a \in \overline{E^\alpha} \cup \overline{E^\beta},\ \forall\ b \in \overline{E^\alpha} \cup \overline{E^\beta},$ *"a ↛ b" ∈* $I_{\alpha\cap\beta}$ *si :*

- $"a \nrightarrow b" \in I_\alpha \wedge "a \nrightarrow b" \in I_\beta$
- $"a \nrightarrow b" \in I_\alpha \wedge a \notin \overline{E^\beta}$
- $"a \nrightarrow b" \in I_\alpha \wedge b \notin \overline{E^\beta}$

Définition 17 $\forall\ a \in \overline{E^\alpha} \cup \overline{E^\beta},\ \forall\ b \in \overline{E^\alpha} \cup \overline{E^\beta},$ *"a ⇌ b" ∈* $I_{\alpha\cap\beta}$ *si :*

- $"a \rightleftharpoons b" \in I_\alpha \wedge "a \rightleftharpoons b" \in I_\beta$
- $"a \rightleftharpoons b" \in I_\alpha \wedge a \notin \overline{E^\beta} \wedge b \notin \overline{E^\beta}$

Lors de la fusion, les définitions ci-dessus sont appliquées pour chaque exécution, pour chaque classe d'invariant et pour chaque couple de types d'événements. Une fois que toutes les exécutions ont été prises en compte, on obtient une liste d'invariants I_G qui contient les invariants satisfaits par l'ensemble des exécutions.

Deuxième étape :

Dans la liste obtenue, certains invariants peuvent être redondants. Dans ce cas, éliminer un invariant de la liste peut alors se faire sans perdre d'information et permet d'avoir une liste plus courte et donc un modèle plus simple. La suppression de ces redondances n'aura aucun impact sur la qualité des détections d'anomalies réalisées à partir du modèle mais permettra très souvent de réduire le temps nécessaire pour évaluer ces invariants lors du processus de détection. Cette étape de réduction est réalisée à la fin de la fusion de toutes les listes intermédiaires d'invariants. En effet, une redondance qui est observable dans une liste intermédiaire I_α ne l'est plus nécessairement au niveau de la liste I_G. Anticiper les réductions en les effectuant au moment de la construction d'une liste intermédiaire d'invariants pourrait donc conduire à une perte d'information.

Pour trouver des invariants redondants et réduire la liste I_G sans risquer de perdre de l'information, il est possible de s'appuyer sur des propriétés de transitivité évidentes :

$$\mathrm{a} \leftarrow \mathrm{b} \wedge \mathrm{b} \leftarrow \mathrm{c} \implies \mathrm{a} \leftarrow \mathrm{c}$$
$$\mathrm{a} \rightarrow \mathrm{b} \wedge \mathrm{b} \rightarrow \mathrm{c} \implies \mathrm{a} \rightarrow \mathrm{c}$$

Les implications exposées ci-dessus identifient un invariant (a $\leftarrow$ c ou a $\rightarrow$ c) qui peut être déduit des deux autres. Il est donc possible de supprimer cet invariant redondant de la liste. Les deux implications caractérisent une propriété de transitivité liée à la classe d'invariants associée ($\leftarrow$ ou $\rightarrow$) : la preuve s'appuie en grande partie sur le fait que la relation de causalité entre les événements d'une même exécution ($\prec_\alpha$ ou $\prec_\beta$ ou ...) est elle-même une relation transitive puisqu'il s'agit d'une relation d'ordre. Cependant, une telle propriété de transitivité n'existe pas dans le cas des deux autres classes d'invariants ($\nrightarrow$ et $\rightleftharpoons$).

$$\text{a} \nrightarrow \text{b} \land \text{b} \nrightarrow \text{c} \;\not\Longrightarrow\; \text{a} \nrightarrow \text{c}$$
$$\text{a} \rightleftharpoons \text{b} \land \text{b} \rightleftharpoons \text{c} \;\not\Longrightarrow\; \text{a} \rightleftharpoons \text{c}$$

Effectuer uniquement des réductions qui correspondent à des redondances réelles permet de ne perdre aucune information mais cela aboutit à une réduction limitée de la taille de la liste d'invariants. Le nombre d'invariants contenus dans la liste dépend en partie de la nature des classes d'invariants adoptées et du nombre de classes d'invariant considérées. Dans nos travaux, nous avons retenu quatre classes alors que d'autres travaux en considèrent moins [SBC+10]. Pour ne pas avoir au final un temps de détection déraisonnable du fait d'une explosion du nombre d'invariants, nous avons cherché à détecter également des redondances probables mais qui ne sont réelles que si des conditions supplémentaires sont vérifiées. Par ailleurs, alors que les deux relations de transitivité présentées précédemment ne concernent à chaque fois qu'une seule classe d'invariants ($\leftarrow$ ou $\rightarrow$), nous allons nous intéresser maintenant à des redondances probables impliquant différentes classes d'invariants.

Pour expliquer la démarche adoptée, nous allons par soucis de simplicité considérer une seule exécution α. Nous allons aussi introduire une notation simplifiée permettant de définir de manière plus condensée des propriétés sur des ensembles d'événements minimaux et maximaux. Si a et b sont deux types d'événements appartenant à $\overline{E}_\alpha$ alors les quatre propriétés suivantes $min(a) \prec min(b)$, $min(a) \prec max(b)$, $max(a) \prec min(b)$ et $max(a) \prec max(b)$ ont les significations suivantes :

$$min(a) \prec min(b) \Longleftrightarrow \forall e \in E^\alpha_{min}(a), \forall f \in E^\alpha_{min}(b), e \prec_\alpha f$$
$$min(a) \prec max(b) \Longleftrightarrow \forall e \in E^\alpha_{min}(a), \forall f \in E^\alpha_{max}(b), e \prec_\alpha f$$
$$max(a) \prec min(b) \Longleftrightarrow \forall e \in E^\alpha_{max}(a), \forall f \in E^\alpha_{min}(b), e \prec_\alpha f$$
$$max(a) \prec max(b) \Longleftrightarrow \forall e \in E^\alpha_{max}(a), \forall f \in E^\alpha_{max}(b), e \prec_\alpha f$$

Notons que ces propriétés, de même que les invariants étudiés, font référence aux ensembles d'événements minimaux et maximaux associés à chaque type d'événement. Par contre, il existe une différence notoire : ces nouvelles propriétés ne reposent que sur

l'utilisation du quantificateur $\forall$ alors que les définitions de certaines classes d'invariant combinent les quantificateurs $\forall$ et $\exists$ (définition 9 et définition 10). Il en résulte les deux implications et les deux équivalences suivantes :

$$
\begin{aligned}
max(a) \prec max(b) &\Longrightarrow a \rightarrow b \\
min(a) \prec min(b) &\Longrightarrow a \leftarrow b \\
max(a) \prec min(b) &\Longleftrightarrow b \nrightarrow a \\
min(a) \prec max(b) &\Longleftrightarrow a \rightleftharpoons b
\end{aligned}
$$

Prenons l'exemple très simple d'une exécution d'une application durant laquelle un processus p_1 exécute un événement de type a puis un événement de type b tandis que le processus p_2 exécute uniquement un événement de type b. Si les événements de type b ne sont pas des événements de communication, ce contre-exemple montre que l'invariant $a \rightarrow b$ (qui est vérifié durant cette exécution) n'implique pas nécessairement que $max(a) \prec max(b)$. En effet, dans cet exemple, l'événement de p_2 est concurrent avec chacun des événements de p_1 ; l'ensemble $max(a)$ contient l'unique événement de type a et l'ensemble $max(b)$ contient les deux événements de type b.

Notons que si les ensembles d'éléments minimaux et maximaux associés aux types d'événements sont des singletons, les deux premières implications deviennent des équivalences. Nous avons déjà signalé que, selon la façon dont les types d'événements sont définis, cette situation peut être fréquente. Pour effectuer des réductions supplémentaires et ainsi améliorer le temps nécessaire à la détection, nous allons donc supposer que les équivalences sont toutes fort probables.

Sous l'hypothèse que les formulations (sous forme de prédicats ou sous forme de relations min/max) sont toutes équivalentes, nous allons considérer tous les ordonnancements de trois types d'événements quelconques. Pour un de ces ordonnancements particulier (par exemple, a puis b puis c), il est possible d'identifier huit relations impliquant les ensembles d'événements minimaux ou maximaux de ces trois types d'événement. Ces huit cas de figure sont détaillés dans la seconde colonne (intitulée relations min/max) du tableau 3.1.

#	relations min/max	réduction envisageable
1	min(a) $\prec$ min(b) $\prec$ min(c)	a $\leftarrow$ b, b $\leftarrow$ c, ~~a $\leftarrow$ c~~
2	min(a) $\prec$ min(b) $\prec$ max(c)	a $\leftarrow$ b, b $\rightleftharpoons$ c, ~~a $\rightleftharpoons$ c~~
3	min(a) $\prec$ max(b) $\prec$ min(c)	a $\rightleftharpoons$ b, c $\not\twoheadrightarrow$ b, ~~a $\leftarrow$ c~~
4	min(a) $\prec$ max(b) $\prec$ max(c)	a $\rightleftharpoons$ b, b $\rightarrow$ c, ~~a $\rightleftharpoons$ c~~
5	max(a) $\prec$ min(b) $\prec$ min(c)	b $\not\twoheadrightarrow$ a, b $\leftarrow$ c, ~~c $\not\twoheadrightarrow$ a~~
6	max(a) $\prec$ min(b) $\prec$ max(c)	b $\not\twoheadrightarrow$ a, b $\rightleftharpoons$ c, ~~a $\rightarrow$ c~~
7	max(a) $\prec$ max(b) $\prec$ min(c)	a $\rightarrow$ b, c $\not\twoheadrightarrow$ b, ~~c $\not\twoheadrightarrow$ a~~
8	max(a) $\prec$ max(b) $\prec$ max(c)	a $\rightarrow$ b, b $\rightarrow$ c, ~~a $\rightarrow$ c~~

TABLEAU 3.1 – Réductions envisagées des invariants

Dans la première ligne du tableau 3.1, la notation $min(a) \prec min(b) \prec min(c)$ signifie que $min(a) \prec min(b)$ et $min(b) \prec min(c)$, ce qui implique que $min(a) \prec min(c)$. Il en va de même pour les sept autres lignes. Aux trois relations min/max associées à une ligne du tableau, nous pouvons donc faire correspondre trois invariants correspondants aux couples de types d'événements (a, b), (b, c) et (a, c) en nous appuyant pour cela sur les équivalences réelles ou probables présentées auparavant. L'idée est alors de supprimer l'invariant associé au couple (a, c) quand les deux invariants associés aux couples (a, b) et (b, c) sont vérifiés. Pour chacun des huit cas, dans la troisième colonne du tableau, la suppression envisagée quand trois prédicats sont vérifiés est indiquée par une barre oblique. Remarquons tout d'abord que les lignes 1 et 8 du tableau correspondent aux simplifications qui sont autorisées par les deux formes de redondance que nous avons vues précédemment (propriété de transitivité liée à la classe d'invariants $\leftarrow$ ou $\rightarrow$). Les lignes 2,4,5 et 7 identifient elles aussi des simplifications qui ne correspondent pas à des redondances dans le cas général mais peuvent être exact dès lors qu'un trois ensembles (d'événements minimaux ou maximaux) est un singleton. Ces quatre lignes correspondent à des simplifications intuitives. Les deux lignes restantes (ligne 3 et ligne 6) correspondent à des simplifications un peu moins évidentes. Elles se caractérisent également par le fait qu'elles mettent en jeu une relation $max() \prec min()$ qui est riche en information. En effet :

$$max(a) \prec min(b) \Longrightarrow max(a) \prec max(b) \wedge min(a) \prec min(b)$$
$$max(a) \prec max(b) \vee min(a) \prec min(b) \Longrightarrow min(a) \prec max(b)$$

Ainsi, à partir de la relation $max(a) \prec min(b)$, on peut déduire l'invariant $b \not\twoheadrightarrow a$ mais aussi les invariants $a \leftarrow b$, $a \rightarrow b$ et $a \rightleftharpoons b$. Nous avons fait le choix d'utiliser en partie cette information pour obtenir de nouvelles règles 3bis et 6bis qui sont plus intuitives et qui sont également plus contraignantes car il faut désormais identifier quatre

invariants dans la liste d'invariants (au lieu de trois) pour pouvoir supprimer l'un d'entre eux.

$$\# \text{ 3bis} \quad a \rightleftharpoons b,\ c \nrightarrow b,\ b \leftarrow c,\ \cancel{a \leftarrow c}$$
$$\# \text{ 6bis} \quad b \nrightarrow a,\ a \rightarrow b,\ b \rightleftharpoons c,\ \cancel{a \rightarrow c}$$

Lors de l'étape de réduction de la liste d'invariants, nous allons donc considérer les 6 relations proposées dans le tableau 3.1 (1, 2, 4, 5, 7 et 8) ainsi que les deux relations modifiées présentées ci-dessus (3bis et 6bis). Les réductions sont appliquées en considérant successivement toutes les classes d'invariant.

3.4 Détection

Une fois le modèle de comportement de l'application calculé, il est possible de l'utiliser pour vérifier que le comportement d'une application (lors d'une nouvelle exécution) est conforme à ce qui est attendu ou, au contraire, détecter d'éventuelles anomalies. Pour rappel, le modèle est dual car il est composé d'un automate et d'une liste d'invariants. La phase de détection est réalisée à l'aide du module *Detect* (voir figure 3.1). Ce module accepte en entrée une trace qui correspond donc à une nouvelle exécution de l'application surveillée. Nous faisons ici l'hypothèse que la trace est évaluée comme étant correcte si elle respecte tous les sous-modèles qui composent le modèle de comportement normal. Au contraire, elle est évaluée comme incorrecte (correspondant à une attaque) si elle n'est pas en conformité avec au moins l'un de ces sous-modèles. Pour évaluer si une trace d'exécution d'une application distribuée est correcte à l'aide de notre approche, nous devons donc effectuer à la fois une vérification par rapport à la liste d'invariants et parcourir l'automate. Cette évaluation peut être réalisée « en ligne » ou « hors ligne ». Dans le contexte de cette thèse, nous avons suivi une approche « hors ligne » pour faciliter le processus d'évaluation. L'approche présentée peut être adaptée pour effectuer de la détection en ligne. Cependant, il faut garder à l'esprit que, comme nous allons le voir par la suite, d'une part la validation d'une trace via un automate peut nécessiter des retours arrières (coûteux en temps) et d'autre part, la violation de certains invariants ne peut parfois être détecté qu'en fin d'exécution.

L'exécution observée, tout comme les exécutions qui ont servi lors de la phase d'apprentissage, est perçue sous la forme d'une trace correspondant à un nouvel ordre partiel. Lors de la phase de détection, plusieurs stratégies sont possibles. Par rapport, à l'ordre partiel, on peut considérer que tous les chemins possibles du treillis associé doivent séparément satisfaire les contraintes liées au modèle ou avoir une attitude plus laxiste et considérer que l'application est correcte dès lors qu'au moins un chemin satisfait ces

contraintes. C'est cette seconde stratégie qui conduit nécessairement à générer moins d'alertes qui a été choisie.

Par rapport à l'automate, deux types de parcours, parcours en profondeur et parcours en largeur, peuvent être adoptés. Un parcours en largeur est plus adapté à une vérification portant sur tous les chemins. De plus, un parcours en largueur est souvent plus consommateur de mémoire (problème combinatoire durant l'exploration de l'automate qui peut être non-déterministe et durant l'exploration de la trace où le choix du prochain événement à consommer se fait souvent entre plusieurs événements concurrents). Puisque nous faisons le choix de valider une trace dès lors qu'un des ordres totaux est compatible avec tous les sous-modèles, le parcours en profondeur est plus adapté. La trace en entrée est consommée (en tenant compte de la relation de causalité) avec l'objectif de construire pas à pas une séquence complète compatible avec l'ordre partiel. Si la séquence, ou un préfixe de cette séquence, ne satisfait pas l'un des sous-modèles, un retour arrière est effectué pour considérer le dernier préfixe où un choix entre plusieurs événements concurrents était possible. Comme nous venons de l'indiquer, il n'est pas nécessaire d'attendre systématiquement qu'une séquence complète soit construite pour éventuellement l'invalider. Dans le cas du modèle à base d'automate, le fait qu'un préfixe n'est pas accepté par l'automate fait que toute extension de ce préfixe ne respectera pas le modèle. Dans le cas de la liste d'invariants, il est parfois possible pour certaines classes d'invariants ($\nrightarrow$ ou $\leftarrow$) de déterminer sur la base d'un préfixe d'une séquence que l'invariant sera nécessairement violé par la séquence complète : nous parlons alors de « faux définitifs ».

Algorithm 8 Detection algorithm

```
1: function DETECT(advance, trace)
2:     end ← true
3:     for i in Dimention do
4:         event ← getEvent(trace, advance, i)
5:         if event != null then
6:             end ← false
7:             if canEvolve(event, advance.getChannel()) then
8:                 nextautstates ← getNextAutStates(event)
9:                 nextinvstates ← getNextInvStates(event)
10:                nextchannel ← getNextChannel(event)
11:                evalstate ← true
12:                if evalOInvstates(nextinvstates) = false then
13:                    evalstate ← false
14:                end if
15:                if evalstate = true then
16:                    for nextstate in nextautstates do
17:                        advancetemp ← Advance(nextstate, nextinvstates,
                               nextchannel, i)
18:                        eval ← DETECT(advancetemp, trace)
19:                        if eval = true then
20:                            return true
21:                        end if
22:                    end for
23:                end if
24:            end if
25:        end if
26:    end for
27:    if end = true then
28:        result ← evalInvStates(advance)
29:        return result
30:    else
31:        return false
32:    end if
33: end function
```

L'algorithme 8 détaille les traitements effectués lors du parcours en profondeur. Il prend en paramètres une trace à évaluer et un niveau d'avancement dans la progression de l'analyse. L'avancement est composé de l'état actuel d'exploration dans l'automate du modèle (autStates), de la liste d'états utilisés pour la vérification des invariants (invStates), de l'état des canaux de communications (Channel) et de l'état dans la consommation de la trace.

De la ligne 3 à la ligne 26, à partir d'un préfixe déjà analysé, l'objectif est de progresser dans l'analyse en étendant ce début de séquence avec un nouvel événement (event). Pour cela, pour chaque processus p_i (ligne 3), le premier événement de p_i qui n'a pas encore été exécuté (au vu de l'avancement courant) est récupéré à l'aide de la méthode *getEvent* (ligne 4). L'événement récupéré est ensuite évalué. Puisque le marqueur *null* figure à la fin de chaque *log*, le test réalisé à la ligne 5 permet de s'assurer que l'événement récupéré est bien le prochain événement non consommé du processus p_i. Dans le cas contraire, la valeur *null* indique que tous les événements de p_i ont déjà été pris en compte : le processus suivant doit alors être considéré. A la ligne 7, l'appel de la méthode *canEvolve* permet de vérifier que cet événement peut effectivement être exécuté à ce stade. Cette méthode est détaillée par la suite. Si l'événement permet d'évoluer dans la reconnaissance, on va successivement calculer les possibles états futurs dans l'automate (ligne 8), les états futurs associés au processus de vérification des invariants (ligne 9) et la mise à jour des canaux de communication. En cas d'appel récursif de la fonction, ces calculs sont nécessaires pour déterminer le futur niveau d'avancement passé en paramètre. Une fois calculés, les états relatifs à des classes d'invariants (telles que $\leftarrow$ ou $\nrightarrow$) peuvent être évalués immédiatement afin de vérifier si certains correspondent à des « faux définitifs » (lignes 11 à 13). Si c'est le cas, l'événement choisi conduit à créer un préfixe de chemin qui ne satisfait pas au moins un des invariants. Continuer l'analyse des séquences qui étendent ce préfixe est inutile : l'appel récursif est donc empêché (ligne 13). Dans le cas contraire, l'événement de p_i peut venir étendre le préfixe déjà connu au moment de l'appel de la fonction. Pour préparer l'appel récursif suivant, il faut alors tenir compte du fait que l'automate n'est pas nécessairement déterministe. Une nouvelle boucle (lignes 16 à 22) permet de lancer un appel récursif pour chaque état de l'automate atteignable après avoir exécuté une transition possible. Ce sont donc bien tous les avancements possibles qui seront ensuite évalués successivement à l'aide d'un appel récursif (ligne 18). Si une de ces évaluations conduit à la construction d'un chemin complet et correct, le processus de détection s'arrête en considérant la trace comme correcte (ligne 19 à 21). Cela signifie que le dernier appel effectué aura conduit à l'exécution des lignes 27 à 29. A ce stade, puisque toute la trace a été consommée (condition d'arrêt de la récursivité), la dernière validation consiste à évaluer l'ensemble des invariants (ligne 28). Notons que les fonctions appelées aux lignes 12 et 28 ne sont pas identiques mais elles concernent toutes les deux la vérification du respect des invariants.

Algorithm 9 algorithme canEvolve

```
1:  function canEvolve(event, advance)
2:      if advance.getAutState().canEvolve(event) then
3:          if event.type() == sending then return true
4:          else if event.type() == internal then return true
5:          else if event.type() == reception then
6:              for e in advance.getChannel() do
7:                  if event.isSentEvent(e) then return true
8:                  end if
9:              end for
10:         end if
11:     end if
12:     return false
13: end function
```

L'algorithme 9 qui décrit la fonction *canEvolve* est similaire à l'algorithme 2 présenté lors de la phase de construction. La différence entre les deux algorithmes est lié au fait que les états de l'automate généralisé ne permet pas de stocker les informations liés aux canaux de communication. L'événement testé est toujours susceptible d'être exécuté à partir de l'état courant s'il s'agit d'un événement d'émission ou d'un événement interne. Par contre, dans le cas d'un événement de réception, l'événement ne peut être exécuté que si l'état courant (notamment la structure de donnée channel) indique que le message qui doit être consommé a bien fait l'objet d'une émission auparavant. Tant que cette contrainte n'est pas respectée, l'événement de réception ne peut pas être pris en compte sans engendrer une violation de la relation d'ordre partiel. Dans le cas d'une application qui se déroule normalement (sans attaque), et à condition que toutes les actions de communication soient observables, la consommation d'un événement de réception peut être retardée durant plusieurs appels de la fonction *canEvolve* mais cette situation ne perdure jamais. Dans la cas d'une exécution perturbée par une attaque, la présence dans la trace d'une réception de message sans émission correspondante est possible. Dans ce cas, la trace ne pourra jamais être entièrement consommée et ceci même si aucune anomalie n'est décelée au niveau des deux modèles utilisés (automate et liste d'invariants). En ce sens, une légère adaptation de cette fonction peut permettre de la transformer à moindre coût en un troisième modèle chargé de vérifier qu'à toute action de réception d'un message correspond une action d'émission de ce message qui peut effectivement toujours avoir lieu avant la réception. Puisque cela permet de différencier des comportements normaux et anormaux, nous présenterons par la suite la vérification de cet invariant de communication basique que nous nommerons « invCom » comme un troisième modèle bien distinct des deux autres.

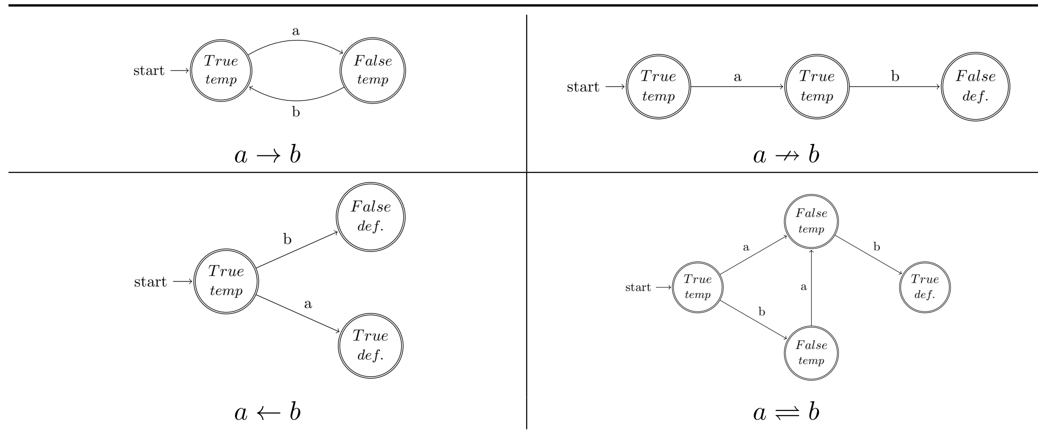

Tableau 3.2 – Évaluation pour les invariants de type a et b

Le tableau 3.4 détaille les automates nécessaires pour la reconnaissance des quatre différentes classes d'invariants. Un automate est nécessaire pour chaque invariant répertorié dans la liste d'invariants. Leurs évolutions sont gérées par la méthode $getNextInvStates$ et ils sont évalués par les méthodes $evalOInvStates$ et $evalInvStates$. Au début de la reconnaissance, la liste des états retenus correspond aux états initiaux de ces automates. Dans chaque automate, les états peuvent etre de quatre sortes : $Truedef.$ pour vrai définitif, $Truetemp$ pour vrai temporaire, $Falsedef.$ pour faux définitif et $Falsetemp$ pour faux temporaire. Comme nous l'avons vu, cette distinction est utile pendant l'évaluation car les automates dont un état a pour valeur $Falsedef.$ vont permettre d'éviter l'exploration en profondeur de chemins forcément incorrects. Cette vérification est effectuée par la méthode $evalOInvStates$ pour les classes d'invariants $\nrightarrow$ et $\leftarrow$. Les autres types d'invariants ne peuvent être évalués définitivement qu'à la fin de l'exécution de la trace à l'aide de la méthode $evalInvStates$.

La complexité de l'algorithme de détection est liée à celle d'un parcours en profondeur (nombre de chemins dans le treillis). Elle se situe donc entre 1 et $\mathcal{S}!/\mathcal{P}$, où $\mathcal{S} = \sum_{i=1}^{n} |E_i^\alpha| = |E^\alpha|$ et $\mathcal{P} = \prod_{i=1}^{n}(|E_i^\alpha|!)$. Cependant d'autres facteurs impactent cette complexité. Elle peut être impactée négativement en cas de non déterminisme de l'automate ou de la trace. En effet, des événements de même type peuvent apparaître sur des processus différents. D'autre part, l'algorithme de généralisation peut, lors de la fusion, provoquer l'apparition d'états ayant deux transitions qui concernent le même type d'événement. La complexité peut aussi être impactée positivement grâce au mécanisme de vérification d'invariants en cours d'exécution à l'aide de la méthode $evalOInvStates$. Ainsi, l'exploration de chemins en profondeur qui sont faux peut être évitée.

3.5 Conclusion

Ce chapitre présente une approche de détection d'intrusion basée sur la modélisation du comportement. L'approche est constituée de deux phases. La première phase vise à apprendre le modèle de comportement d'une application distribuée. La deuxième phase consiste à utiliser le modèle de comportement normal pour détecter les déviations par rapport à ce comportement.

L'idée est de présenter les différentes étapes de l'approche de modélisation : le passage de fichiers de logs d'une exécution à un modèle de comportement d'une exécution ; la modélisation du comportement de l'application à l'aide de plusieurs modèles d'exécutions). Ensuite, une approche de détection reposant sur les différents sous-modèles (automate et invariants) a été proposée. Ce chapitre a permis de présenter différents types de modèles et leurs caractéristiques ainsi que de discuter des choix retenus lors des phases de construction et de détection. Enfin, des calculs de complexité ont été réalisés pour différentes étapes de l'approche. Ces derniers permettent de soulever les limites de l'approche et d'identifier les données qui ont une influence sur les différentes complexités.

Les modélisations à base d'automates et de listes d'invariants avaient fait l'objet de différentes études. Notre contribution a porté sur la définition d'une approche commune pour les deux modèles (treillis des états globaux, événements minimaux et maximaux par type d'événement, états globaux particuliers, ...) et sur une caractérisation rigoureuse de quatre classes d'invariants en identifiant formellement les relations qui les unissent.

Chapitre 4

Passage à l'échelle

Le chapitre précédent traitait de l'approche proposée pour inférer un modèle de comportement à partir de logs d'une application distribuée. Ce modèle est utilisé dans le contexte de la détection d'intrusion. Ce chapitre se focalise sur les problèmes de passage à l'échelle et sur différentes propositions permettant de traiter ces problèmes. Pour cela, on propose dans un premier temps d'évoquer les problèmes connus ainsi que les impacts sur notre approche. Ensuite, deux propositions permettant de passer à l'échelle seront proposées.

4.1 Problèmes de passage à l'échelle

L'approche de modélisation et de détection détaillée dans le chapitre précédent a été présenté en précisant les complexités de différentes étapes de l'approche. Parmi ces complexités, nous avons identifié deux problèmes pouvant empêcher un passage à l'échelle. Le premier problème concerne la croissance exponentielle du treillis des états globaux cohérents. Le second problème concerne la généralisation d'automates avec un nombre d'états élevé. Ces deux problèmes sont traités dans les deux sous-sections suivantes.

4.1.1 La croissance exponentielle du treillis

Le treillis des états globaux cohérents est une structure algébrique qui contient l'ensemble des coupures cohérentes d'une exécution (Voir Section 2.2). Ce type de structure peut croître de façon exponentielle [Gar12a]. Dans la littérature, ce problème est connu comme le « State Explosion Problem » [Val96]. Il est particulièrement abordé dans le domaine de l'analyse et de la vérification de comportements de systèmes distribuées [FG05]. En pratique, ce problème impacte tous les travaux cherchant à énumérer ou analyser l'ensemble des états globaux cohérents que le système peut atteindre. Lors

de cette énumération et ce même pour des systèmes relativement petits et des exécutions d'une durée réduite, le nombre d'états est souvent trop grand pour être géré.

Dans notre solution, ce problème impacte négativement le temps de calcul et la place mémoire nécessaires à différentes étapes de l'approche qui utilisent le treillis ou des structures liées au treillis (automates intermédiaires). Pour toutes les exécutions, la phase de construction du modèle proposée nécessite d'énumérer l'ensemble des états du treillis et des transitions afin de générer l'automate intermédiaire. En parallèle, cette énumération est également utilisée pour la première phase de calcul des invariants intermédiaires (mise à jour des ensembles d'événements minimaux et maximaux à chaque nouvelle transition). La phase de fusion et généralisation considère ensuite l'ensemble des automates intermédiaires, chacun ayant une structure qui correspond à celle d'un treillis. La fusion qui vise à obtenir un seul automate n'est pas trop impactée. Par contre, lors de la généralisation, tous les k futurs des états de l'automate fusionné sont comparés aux autres k futurs des autres états, ce qui peut présenter un coût en calcul considérable. La phase de construction du modèle est donc fortement impactée par ce problème. Lors de la phase de détection un parcours en profondeur est effectué pour évaluer les chemins dans la trace. Ce parcours est couplé avec l'utilisation des invariants pour éviter que certains chemins incorrects soient explorés en profondeur. L'impact que le « State Explosion Problem » a sur cette phase de détection existe aussi mais certains invariants permettent parfois de réduire l'ensemble des chemins parcourus.

Des approches cherchent à atténuer les effets du « State Explosion Problem ». Pour cela, des méthodes vont utiliser une abstraction de l'ensemble d'état [FG05]. D'autres vont considérer un sous-ensemble d'états pour limiter l'impact du problème [GCB18, YACK11]. Enfin, d'autres types de treillis peuvent être considérés [Gar12a]. Par exemple, dans les articles [GCB18, YACK11], les seuls états retenus correspondent aux « ground states ». Ces états particuliers (qui sont parfois peu nombreux) correspondent aux coupures dans lesquelles il n'y a pas de messages en transit. Ils sont utilisés dans ces articles pour simplifier le calcul des invariants. Cependant, ils répondent à un objectif précis. Ils ne peuvent pas être utilisés pour établir un modèle comportant des séquences d'événements comme les automates.

Notre approche nécessite à la fois une construction d'automates et d'invariants. Il est donc nécessaire d'utiliser une méthode de réduction de l'ensemble d'états qui permette de générer ces deux types de modèles.

4.1.2 La généralisation d'automates de grande taille

Dans les approches qui cherchent à construire un modèle de comportement à l'aide de traces d'exécutions [THH+16, BBEK14, LMP08], la construction du modèle se divise en

deux étapes. Une première étape consiste à récupérer les traces de plusieurs exécutions pour obtenir un modèle de comportement global de l'application. Une seconde étape consiste à généraliser ce modèle à l'aide d'un algorithme comme k-tail pour introduire de nouveaux comportements.

L'utilisation de l'algorithme k-tail pour généraliser un automate nécessite une comparaison entre les k-futurs des différents états. Ces comparaisons ont pour but de déterminer quels états doivent être fusionnés. Dans la section 3.3.1, nous avons exposé une approche permettant de généraliser un automate à l'aide de k-tail. Pour rappel, deux algorithmes ont alors été présentés. L'algorithme 6 identifie dans un premier temps des couples d'états à fusionner. Il présente une complexité qui est en $\mathcal{O}(\mathcal{E} \times (\mathcal{E} - 1))/2)$ où $\mathcal{E}$ correspond au nombre total d'états de l'automate avant généralisation. L'algorithme 7 fusionne ensuite les couples d'états identifiés comme étant similaires. Dans le pire des cas, toutes les paires d'états sont concernées. Les complexités de ces deux algorithmes ont donc un lien direct avec le nombre d'états présents dans l'automate à généraliser.

L'application d'un algorithme de généralisation comme k-tail pour un automate de grande taille peut donc poser problème. Dans le contexte de nos travaux, une taille importante peut être causée par :

- l'apprentissage d'un modèle de comportement à l'aide d'un nombre important d'exécutions ;
- le fait que l'automate construit à l'aide des traces d'exécution a une structure similaire au treillis des états globaux cohérents (voir section 4.1.1) et donc une taille potentiellement importante (fonction du nombre de processus et de la durée des exécutions).

Il est donc nécessaire de faire des propositions pour limiter l'impact lié au nombre important d'états sur le processus de généralisation.

4.2 Calcul d'un sous ensemble du treillis

L'objectif de cette proposition est de traiter le problème de croissance exponentielle du treillis exposé en 4.1.1 et, par extension, de résoudre partiellement le problème de généralisation exposé en 4.1.2. Cette proposition, tout comme les propositions dans l'état de l'art [FG05, YACK11, GCB18], vise à réduire le nombre total d'états et de transitions en calculant un sous-ensemble du treillis.

Dans le contexte de ce travail, nous avons besoin de conserver des séquences d'événements pour pouvoir générer ensuite un automate (voir section 3.2.2) : nous allons donc chercher à sélectionner des chemins complets dans le treillis. Le fait d'ignorer certains

chemins du treillis nous permet de réduire le nombre d'état et le nombre de transitions qui composent la nouvelle structure. Cependant, nous avons aussi besoin de conserver un sous ensemble du treillis nous permettant toujours de raisonner sur des états particuliers de ce treillis (« join irreducible » ou « meet irreducible ») pour calculer des invariants (voir section 3.2.3). Ceci impose que les états indispensables à ce calcul soient conservés et figurent au sein des chemins sélectionnés. Au final, le sous-ensemble conservé doit être suffisamment riche pour que la liste d'invariants générée soit aussi précise qu'avec le treillis complet.

Pour sélectionner les chemins qui seront retenus, nous utilisons des heuristiques. Ces heuristiques peuvent être de différentes natures (sélection aléatoire des chemins, priorité donnée à certains types d'événements, ordre de priorité sur les processus). Selon la stratégie adoptée, l'objectif de conserver l'ensemble des états « join irreducible » ou « meet irreducible » ne sera pas forcement atteint. Ainsi, à l'évidence, un choix aléatoire des chemins conservés ne permet d'avoir aucune garantie complémentaire. Pour définir une heuristique qui permette d'atteindre cet objectif, il est nécessaire de comprendre dans quelles séquences ces états peuvent apparaître. Un état « join irreducible » couvre exactement un autre état. Autrement dit, l'état n'a qu'un seul état prédécesseur direct. L'heuristique doit donc permettre de conserver au moins un chemin intégrant cette transition couvrante. Cet objectif peut être atteint si la stratégie de sélection retenue donne une priorité (lors de la définition du chemin sélectionné) à l'axe correspondant à cette transition couvrante et donc cherche à consommer le plus rapidement possible l'événement qui permet d'atteindre l'état « join irreducible ». Dans le treillis des états globaux cohérents, chaque axe est associé à un processus. Supposons que l'axe couvrant soit associé au processus p_i. L'exploration d'un chemin avec comme heuristique « p_i est prioritaire par rapport à tous les autres processus » peut sembler appropriée pour obtenir l'ensemble des états « join irreducible » liés à p_i. Cependant, des événements de p_i peuvent être causés par des événements d'autres processus et établir une séquence minimale pour atteindre chaque état « join irreducible » est complexe dans un parcours ordonné.

Heureusement, dans le cas des états « meet irreducible », le problème est plus simple à gérer. Les états « meet irreducible » d'un axe apparaissent quand le processus correspondant à cet axe est moins prioritaire que tout les autres processus. Autrement dit, l'événement lié à l'état doit être consommé au moment où il est le dernier à pouvoir être consommé. Des priorités statiques entre les n processus peuvent être utilisées pour déterminer ces séquences particulières qui contiennent des états « meet irreducible ». On utilise la notation $p_i \rhd p_j$ pour indiquer que p_i est prioritaire sur p_j. Parmi n processus, $n!$ permutations peuvent être définies. Chaque permutation est une règle de sélection

qui identifie une seule séquence dans le treillis complet : étant donné un préfixe de la séquence sélectionnée (c'est-à-dire une coupure cohérente), la règle identifie le prochain événement de la trace qui étendra ce préfixe pour obtenir la prochaine coupure cohérente et respectera l'ordre de priorité entre processus fixé par la règle. Cette stratégie déterministe sélectionne uniquement des séquences dans la coque concave. Une heuristique de priorité impliquant n processus peut s'écrire sous la forme « $p_{x_1} \rhd p_{x_2} \rhd \ldots \rhd p_{x_n}$ ». Dans cette heuristique, le processus p_{x_n} est le moins prioritaire et p_{x_1} le plus prioritaire.

Algorithm 10 algorithme de construction d'un chemin

```
 1: function PATHCONSTRUCT(state, trace, heuristic)
 2:     for i in heuristic do
 3:         event ← trace.getEvent(state, heuristic.get(i))
 4:         if canEvolve(state, event) then
 5:             newstate ← State(state, heuristic.get(i), event)
 6:             newtrans ← Transition(state, newstate, event)
 7:             updateMinMax(newtrans)
 8:             transitionset.add(newtrans)
 9:             PATHCONSTRUCT(newstate, trace, heuristic)
10:             return
11:         end if
12:     end for
13: end function
```

L'algorithme 10 détaille le fonctionnement de l'exploration d'un chemin à l'aide d'une heuristique de priorité. Cet algorithme est similaire à l'algorithme 1 qui avait pour objectif de construire le treillis des états globaux cohérents. Ici, on note l'apparition d'un paramètre heuristique qui correspond à une liste de priorités pour l'exploration d'un chemin. Elle est sous la forme « $p_{x_1} \rhd p_{x_2} \rhd \ldots \rhd p_{x_n}$ » que nous venons de décrire : l'exécution de la ligne 2 permet de considérer les n processus dans l'ordre de priorité défini. Une autre différence avec l'algorithme cherchant à construire le treillis complet apparaît à la ligne 10. L'instruction *return* permet d'éviter l'exploration de plusieurs chemins dans le treillis.

Pour obtenir l'ensemble des états « meet irreducible » (pour tous les processus), au plus n heuristiques de priorités sont nécessaires (parmi les $n!$ possibles). Ces heuristiques doivent contenir à chaque fois un processus différent qui sera le moins prioritaire. La construction des n chemins du treillis permet d'obtenir une structure qui est un sous-ensemble du treillis des états globaux cohérents. A noter que la définition des n heuristiques peut se faire très simplement en procédant à des permutations circulaires

qui vont générer n heuristiques respectant la contrainte d'avoir un processus moins prioritaire différent :
« $p_1 \rhd p_2 \rhd ... p_{n-1} \rhd p_n$ », « $p_2 \rhd p_3 \rhd ... p_n \rhd p_1$ », ... et « $p_n \rhd p_1 \rhd ... p_{n-2} \rhd p_{n-1}$ ».

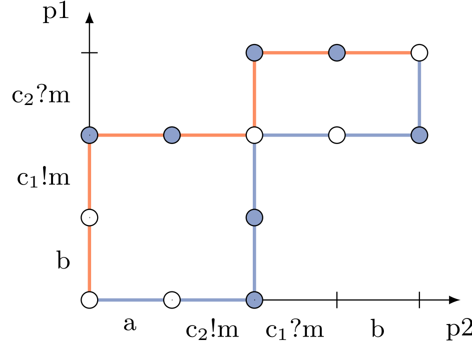

FIGURE 4.1 – Deux séquences sélectionnées avec $p_1 \rhd p_2$ (en rouge) et $p_2 \rhd p_1$ (en bleu)

Si on considère l'exemple de l'exécution α exposé dans la section 2.2, on obtient à l'aide de l'approche proposée la structure exposée dans la figure 4.1. Pour construire cette structure, deux heuristiques ont été utilisées. L'heuristique $p_1 \rhd p_2$ (en rouge) et l'heuristique $p_2 \rhd p_1$ (en bleu) ont permis d'obtenir respectivement les séquences d'événements $<b; c_1!m; a; c_2!m; c_2?m; c_1?m; b>$ et $<a; c_2!m; b; c_1!m; c_1?m; b; c_2?m>$. Elles permettent aussi d'obtenir les états « meet irreducible » (états en bleu) du processus le moins prioritaire. Ainsi, pour l'heuristique $p_1 \rhd p_2$, on a obtenu les états « meet irreducible » qui correspondent au processus p_2. Ils ont comme coordonnées : a :(0,2), $c_2!m$:(1,2), $c1?m$:(2,3) et b :(3,3). Les deux séquences sélectionnées à l'aide des heuristiques partagent trois coupures cohérentes : l'état global initial en (0,0), l'état global final en (4,3) et l'état de coordonnées (2,2). Ainsi, nous avons au final quatre chemins : aux chemins bleu et rouge s'ajoutent deux séquences valides supplémentaires également présentes dans le treillis complet : $<b; c_1!m; a; c_2!m; c_1?m; b; c_2?m>$ et $<a; c_2!m; b; c_1!m; c_2?m; c_1?m; b>$. De manière générale, le processus de sélection fondé sur l'utilisation de n heuristiques peut conduire à sélectionner un nombre de chemins différents qui peut être inférieur ou supérieur à n.

L'approche de sélection de séquences proposée a une complexité temporelle en $\mathcal{O}(n \times \mathcal{S})$ ou n correspond au nombre de processus (ou nombre d'heuristiques) et $\mathcal{S}$ est la longueur de n'importe quel chemin du treillis (voir la section 3.2.1). Elle est donc supérieure de n fois à la complexité minimale de la construction du treillis (en cas de faible concurrence entre événements) mais très inférieure à la complexité maximale de cette construction (en cas de forte concurrence entre événements).

Ne plus considérer la structure initiale (un treillis complet) mais cette nouvelle structure (généralement plus petite) n'engendre pas de changements méthodologiques dans les étapes suivantes : les algorithmes ne changent pas mais leur exécution est moins coûteuse en temps et en mémoire. Au niveau de la construction d'un modèle intermédiaire, le calcul des invariants reste inchangé. En effet, les états « meet irreducible » qui sont nécessaires à leur calculs sont conservés. L'automate quand à lui est composé de moins d'états et de transitions. La modification de la structure de l'automate impacte aussi l'étape de fusion et de généralisation. Moins d'états étant présents dans les automates intermédiaires, le processus de généralisation avec un algorithme comme k-tail va comparer moins d'états (leurs k-futurs) entre eux. Le processus de généralisation va donc être plus rapide. L'automate généralisé sera cependant différent. Cette différence aura un impact sur les résultats de détection au niveau de l'automate. Une évaluation de l'approche en terme de passage à l'échelle et de qualité du modèle sera présentée dans le chapitre 6.

4.3 Fusion et généralisation itérative

Dans cette section, nous proposons une approche pour limiter le problème de passage à l'échelle lié à la généralisation d'automates de grande taille à l'aide d'un algorithme comme k-tail (voir section 4.1.2). Le problème est du au nombre important d'états à comparer suite à une fusion globale d'automates.

Premier constat, dans le processus d'apprentissage, les exécutions sont prises en compte indépendamment les unes des autres. Les modèles intermédiaires (plus précisément les automates intermédiaires) sont donc des entités distinctes. Dans l'approche de fusion et de généralisation que nous qualifions de globale, les opérations de fusion et de généralisation sont exécutées chacune une seule et unique fois. Dans un premier temps, les automates sont tous fusionnés pour obtenir un unique automate qui est ensuite généralisé dans une seconde étape.

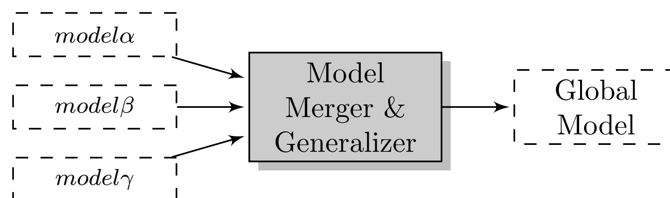

FIGURE 4.2 – Processus de fusion et généralisation global pour 3 modèles intermédiaires

La figure 4.2 détaille le processus de fusion et de généralisation global pour les modèles intermédiaires liés aux exécutions α, β et γ.

Nous conservons le même objectif que dans l'approche de fusion et de généralisation globale. Le but est d'obtenir un unique automate qui représente le fonctionnement normal de l'application. Cet automate doit respecter une contrainte forte : il doit accepter l'ensemble des comportements appris lors des différentes exécutions de la phase d'apprentissage. L'automate obtenu doit aussi (sans nécessairement que cela soit précisément quantifier) pouvoir accepter de nouveaux comportements et avoir une taille raisonnable. À ces attendus, qui sont aussi ceux de l'approche dite globale, nous souhaitons en plus réduire le temps nécessaire pour passer d'un ensemble d'automates intermédiaires à un automate généralisé. L'utilisation de la solution k-tail donnant des résultats satisfaisants (acceptation des comportements appris, introduction de nouveaux comportements acceptés, réduction de la taille de l'automate), nous souhaitons par ailleurs conserver cette brique de base dont l'inconvénient principal reste qu'elle ne permet pas de passer à l'échelle.

Partant du constat que l'exécution d'un algorithme comme k-tail permet de prendre en entrée un automate et de produire en sortie un automate de taille plus réduite (en fusionnant les états qui ont les mêmes futurs de longueur k), nous avons envisagé d'effectuer les fusions de façon progressive et d'appliquer le processus de généralisation à plusieurs reprises mais en espérant travailler à chaque fois sur un automate en entrée qui reste de taille raisonnable. Ainsi, une solution en deux étapes peut être adoptée. Dans une première étape, les automates intermédiaires appris sont divisés en plusieurs sous-ensembles. Dans chaque sous-ensemble, les automates présents sont fusionnés et généralisés dans l'optique d'avoir un automate généralisé par sous-ensemble. Chacun de ces nouveaux automates comportera moins d'états que la fusion de l'ensemble des automates du sous-ensemble dont il est issu. Dans une seconde étape, une fusion et une généralisation de ces nouveaux automates permet de construire un unique automate global et généralisé. Bien évidement, le nombre de phases peut être augmenté en ajoutant autant de phases intermédiaires que nécessaire.

Dans cette approche, nous souhaitons appliquer le processus de fusion et généralisation sur des automates comportant toujours un nombre d'états raisonnable. Ces processus de généralisation vont donc effectuer moins de comparaisons et générer des automates plus petits. Ces nouveaux automates seront à nouveau utilisés lors d'autres phases de fusion et généralisation. Pour que l'approche ait un intérêt, les automates doivent donc être suffisamment généralisés pour limiter le nombre de comparaisons nécessaire dans les phases de généralisation futures. Notre pari est le suivant : effectuer plusieurs fois peu de comparaisons doit s'avérer moins coûteux qu'effectuer une seule fois beaucoup de comparaisons.

Pour fusionner et généraliser plusieurs automates intermédiaires, plusieurs stratégies peuvent être envisagées. Nous en avons déjà présenté une mais de nombreuses autres solutions sont possibles. Par exemple, les modèles intermédiaires correspondant à des exécutions pourraient être fusionnés et généralisés deux à deux puis les modèles de ce nouveau niveau pourrait à nouveau être fusionnés et généralisés deux à deux jusqu'à obtenir un modèle unique. Une autre approche (celle présentée ci-dessous dans la figure 4.3) serait de fusionner et de généraliser itérativement l'automate en cours de construction avec l'automate issue d'une nouvelle trace. Parmi ces diverses stratégies, certaines peuvent être plus performantes ou précises que d'autres (ceci variant probablement selon la nature des exécutions analysées). Cependant, notre objectif ici n'est pas de comparer ces différentes stratégies mais de voir si ce type d'approche permet de passer à l'échelle.

Dans le contexte de cette thèse nous proposons et étudions une approche de généralisation itérative. Une stratégie itérative a comme avantage de générer le modèle progressivement à l'aide de modèles intermédiaires. Cette stratégie permet donc de mettre à jour le modèle global sans avoir besoin de fusionner l'ensemble des traces et de les généraliser à nouveau.

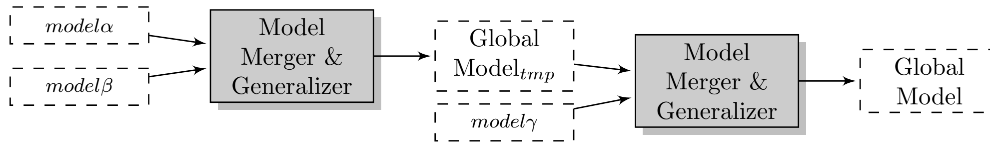

FIGURE 4.3 – Processus de fusion itératif pour 3 modèles intermédiaires

La figure 4.3 présente le fonctionnement de la généralisation itérative pour les modèles intermédiaires. Dans cette figure, le processus de fusion concerne 3 modèles intermédiaires. Deux modèles intermédiaires correspondant aux exécutions α et β sont fusionnées et généralisés lors d'une première étape ce qui nous permet d'obtenir un modèle temporaire $GlobalModel_{tmp}$ pour ces deux exécutions. Ce modèle est ensuite fusionné et généralisé lors d'une deuxième étape avec un modèle correspondant à l'exécution γ pour donner le modèle global $GlobalModel$.

La généralisation selon une approche itérative présente des caractéristiques communes avec l'approche de généralisation globale. Dans les deux cas, les modèles obtenus acceptent l'intégralité des traces utilisées pour les construire. D'autre part, les automates étant généralisés, ils acceptent plus de comportements que ceux utilisés pour les apprendre. Cependant, les modèles (automates) obtenus à la fin de la phase de construction ne sont pas les mêmes. Le modèle obtenu via notre approche est impacté par l'ordre

de prise en compte des traces ce qui n'est pas le cas pour l'approche traditionnelle : deux ordres de prise en compte différents donnent très probablement deux automates finaux différents.

L'évaluation théorique de l'approche est dépendante de l'algorithme de généralisation, de la structure des automates et de paramètres de généralisation (k pour k-tail). Évaluer son impact à priori semble donc complexe. La proposition sera évaluée à posteriori en terme de passage à l'échelle et de qualité de détection dans le chapitre 6.

4.4 Conclusion

Ce chapitre présente des problèmes de passage à l'échelle et des propositions pour les traiter. Il expose dans un premier temps un problème connu dans la modélisation de systèmes distribués (croissance exponentielle du treillis des états globaux cohérents) et son impact sur l'approche proposée pour construire et exploiter un modèle. De plus, il traite un problème lié à l'approche de généralisation et à notre contexte (généralisation d'automates de grande taille). Dans un deuxième temps, une proposition visant à adresser le premier problème est exposée. Cette proposition consiste à construire un sous ensemble du treillis à l'aide de n heuristiques permettant d'explorer un nombre fini de chemins. Dans un troisième et dernier temps, une proposition qui cherche à réduire le problème lié à la généralisation d'automates de grande taille est détaillée. La proposition repose sur la fusion et la généralisation progressive des automates.

Les propositions détaillées dans ce chapitre ont pour objectifs de résoudre les problèmes de passage à l'échelle, de ne pas nécessiter un changement du processus de génération d'automates ou d'invariants et de ne pas avoir d'impact négatifs sur le modèle (les résultats de détection).

Chapitre 5

Évaluation

Ce chapitre propose une évaluation de l'approche de modélisation proposée dans le contexte de la détection d'intrusion. Dans un premier temps, nous détaillons deux cas d'études représentatifs d'applications distribuées qui sont utilisées pour évaluer l'approche. Par la suite, une évaluation des performances est réalisée. Enfin, une évaluation sur la qualité du modèle et sur les capacités de détection est réalisée.

5.1 Cas d'étude n° 1 : Une application de commerce en ligne

Une application d'e-commerce basique est considérée comme premier cas d'étude. Cette application jouet identifie deux type d'entité (un serveur et un client) et n'accepte que deux types de comportement avec un nombre fixe d'échanges de messages. Les différents échanges entre un client et un serveur pour ces deux comportements sont détaillés dans la figure 5.1. L'échange de messages de gauche correspond à l'achat d'un produit par un client auprès d'un serveur distant. L'échange de messages à droite correspond à l'échec dans la recherche d'un produit auprès du serveur.

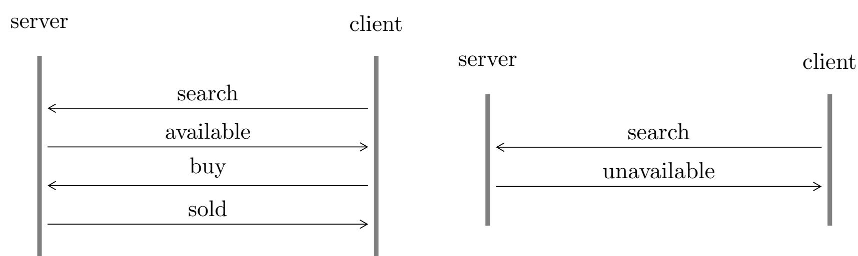

FIGURE 5.1 – Échanges entre le client et le serveur : achat (gauche), échec (droite)

Pour effectuer nos expérimentations, nous considérons une architecture composée de trois machines comprenant deux clients et un serveur et de deux scénarios.

Le premier scénario considère que deux clients vont effectuer des achats auprès d'un serveur lors d'une seule et même exécution. Ces deux clients et le serveur vont réaliser la suite d'échange achat (gauche) décrite dans la figure 5.1. Un ordre total possible pour ce scénario est décrit dans la figure 5.2.

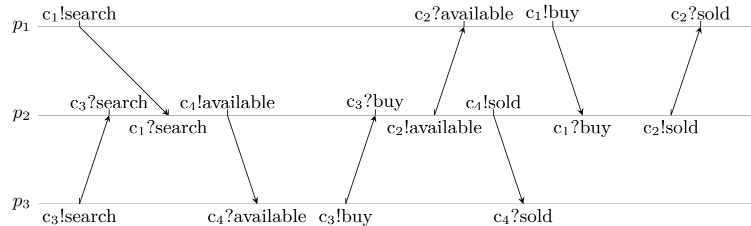

FIGURE 5.2 – Exemple d'ordre total possible pour le premier scénario

Lors de chaque exécution, un client exécute quatre événements tandis que le serveur en exécute systématiquement huit. Huit événements concernent le premier client ou l'interaction entre le premier client et le serveur : ces événements apparaissent toujours dans le même ordre. De même, huit événements concernent le second client ou l'interaction entre le second client et le serveur : ces événements apparaissent aussi toujours dans le même ordre. L'indéterminisme vient du fait que les deux catégories d'événements que nous venons de mettre en avant peuvent s'entrelacer n'importe comment. Chaque séquence complète contient 16 événements et un entrelacement est caractérisé (par exemple) par les huit positions des événements concernant le premier client ou l'interaction entre ce premier client et le serveur. Il y a donc au total 12870 entrelacements possibles qui correspondent au choix de 8 positions parmi 16 positions possibles. L'avantage de cet exemple simple est qu'il permet d'avoir un nombre fini d'ordres totaux possibles mais aussi un nombre fini d'ordres partiels pouvant être observé. Puisque les deux séquences de quatre actions exécutées par les clients sont fixes, seule la dernière séquence correspondant à l'activité du serveur varie et engendre des ordres partiels différents. Pour évaluer notre approche, nous avons collecté 70 traces qui correspondent aux 70 ordres partiels possibles. Pour calculer le nombre d'ordres partiels différents, il faut cette fois considérer la séquence des huit événements du serveur. Les 70 possibilités correspondent au choix de 4 positions (pour les événements en lien avec un client donné) parmi 8 positions possibles.

Le second scénario envisagé implique les mêmes clients. Dans ce scénario, un client va effectuer un achat auprès du serveur alors que le deuxième va échouer systématiquement

dans la recherche d'un produit. Ces deux clients et le serveur vont réaliser les suites d'échanges achat (gauche) et échec (droite) décrites dans la figure 5.1. Un ordre total possible pour ce scénario est décrit dans la figure 5.3.

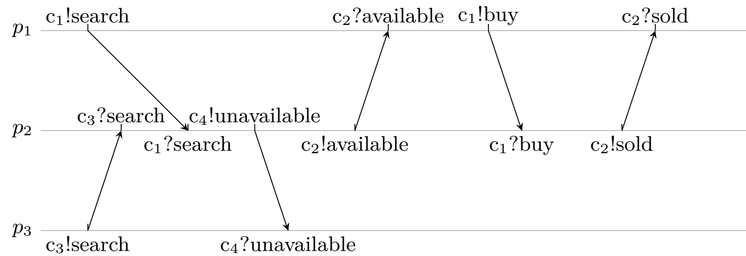

FIGURE 5.3 – Exemple d'ordre total possible pour le second scénario

Le nombre d'ordres totaux possibles dans ce second scénario est de 495. Suivant la même logique, 15 traces qui correspondent aux 15 ordres partiels possibles ont été retenues.

Ce cas d'étude contient relativement peu d'événements et de comportements. Il a été utilisé pour évaluer la validité des implémentations liées aux différentes étapes de l'approche. Il permettra dans la section 5.4.2 de procéder à une évaluation de la complétude d'un modèle généralisé.

5.2 Cas d'étude n° 2 : Un système de fichiers distribué

5.2.1 Présentation

Le deuxième cas d'étude considéré est l'application XtreemFS [CCF+07]. XtreemFS est un système de fichiers distribué tolérant aux fautes. Il peut être réparti sur plusieurs machines. Le design de XtreemFS est basé sur une architecture object-based storage. Le terme objet signifie qu'un fichier peut être découpé en parties de même taille appelés objets. Ces objets sont répartis entre les serveurs de stockage alors que les méta-données des fichiers sont stockées séparément sur un serveur de méta-données. L'utilisation du point de vue de l'utilisateur est similaire à celle d'un système de fichiers local, à la différence que les processus utilisateurs cherchant à accéder au système de fichiers vont passer par le logiciel Filesystem in UserSpacE (FUSE) comme décrit dans la figure 5.4. Le schéma de cette figure est directement extrait du chapitre 2.4 intitulé "Architecture" du guide d'installation et d'utilisation de XtreemFS (`http://www.xtreemfs.org/xtfs-guide-1.4/index.html`).

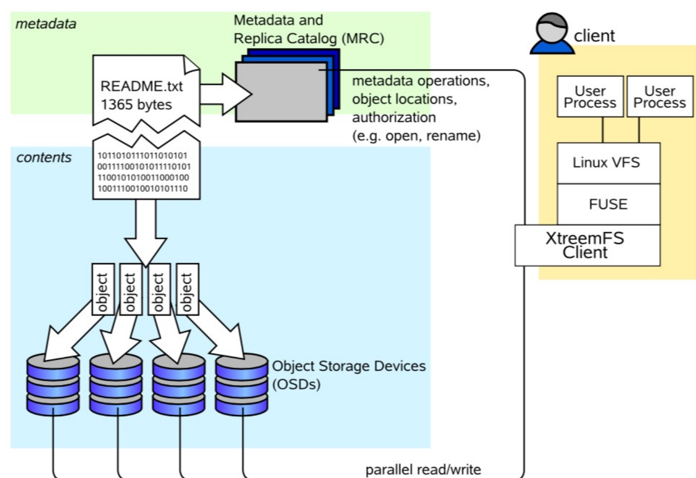

FIGURE 5.4 – Architecture d'XtreemFS

Pour répondre aux besoins de ce système, XtreemFS propose une architecture dans laquelle interviennent 4 composants :

— Le Metadata and Replication Catalog (MRC) stocke les métadonnées des fichiers tels que le nom, la taille ou les permissions. Plusieurs instances de MRC peuvent tourner sur différents serveurs. Le MRC est responsable de l'authentification et des gestions d'accès pour les utilisateurs.

— L'Object Storage Device (OSD) est chargé du stockage du contenu des fichiers. Il peut utiliser une technique appellée striping pour séparer le contenu du fichier en objets. Ces objets sont distribués parmi les OSD présents sur le réseau. Cela permet aux clients de lire et d'écrire les objets en parallèle pour augmenter la bande passante. XtreemFS supporte différentes politiques de striping qui peuvent être spécifiées pour les fichiers.

— Le Directory Service (DIR) est utilisé comme une base de donnée centrale qui fait le lien entre tous les services d'XtreemFS. Les instances de MRC utilisent le DIR pour obtenir les adresses IP des OSD disponibles. Les utilisateurs indiquent quel DIR utiliser quand ils souhaitent monter un volume XtreemFS.

— Le Client permet l'accès aux services d'XtreemFS. Pour celà, il dispose de plusieurs applications pour la gestion des volumes (xtfs mkvol, xtfs rmvol, xtfs mrcdbtool, ...) et leur utilisation (xtfs mount, xtfs umount, ...).

5.2.2 Attaques contre le système de fichiers distribué

Pour évaluer l'approche de détection, Il est nécessaire d'avoir des traces contenant des attaques contre une application distribuée. Pour cela, nous avons étudié le système de fichiers et réalisé des attaques sur une version non sécurisée de celui-ci. Ainsi, cinq attaques ont été découvertes et mises en œuvre. Ces attaques ont permis de violer à la fois la disponibilité, l'intégrité et la confidentialité des données sur le système de fichiers XtreemFS.

5.2.2.1 Attaque via le changement d'IP d'un OSD

L'objectif de cette première attaque est de faire croire au DIR que l'adresse IP d'un OSD a été modifiée. Ainsi, lorsque le DIR communiquera l'adresse IP de l'OSD au Client, cette adresse sera fausse, et le Client n'arrivera pas à se connecter.

Pour générer cette attaque, il a été nécessaire d'observer les échanges entre les différents composants d'XtreemFS lors de l'expérience suivante. L'adresse IP de l'OSD a dans un premier temps été modifiée. L'OSD a ensuite été relancé grâce au client à l'aide de la commande (xtreemfs-osd restart). Les échanges suivants ont été observés entre le DIR et l'OSD, ces derniers se composent de 5 séries de 3 messages. Le descriptif des échanges à l'allumage d'un OSD est détaillé dans la figure 5.5.

Les 5 séries de messages envoyées par l'OSD au DIR ont tout d'abord été rejouée. Cependant, nous avons fait face à quelques problèmes. Lors d'un premier rejeu, la troisième série d'échange a été refusée par le DIR. Le changement d'adresse IP de l'OSD n'a alors pas été pris en compte. Le refus de cette série de messages était lié à un problème de version dans le message. Une fois que les versions présentes dans les messages composant cette série ont été modifiées, l'attaque a été de nouveau rejouée. Ce nouveau rejeu a mené à l'apparition d'une exception de type "ConcurrentModificationException" lors de la cinquième série de messages et à un changement d'adresse IP non pris en compte par le DIR. Cette exception était aussi liée à un problème de version incorrecte présente dans les messages. Enfin, après avoir patché l'ensemble des messages dans les troisièmes et cinquièmes séries de messages, il a été possible de modifier l'adresse IP de l'OSD auprès du DIR sans que celle-ci ait été réellement changée.

5.2.2.2 Attaque sur la création de fichiers

La seconde attaque avait pour objectif de mettre le système dans un état incohérent à l'aide de la création de fichier. Pour cela, les échanges entre le client et les différents éléments du système ont été observés. Ainsi, lorsqu'un client crée un fichier sur le système, des paquets sont échangés avec le MRC, le DIR puis l'OSD, comme sur la figure 5.6. Une

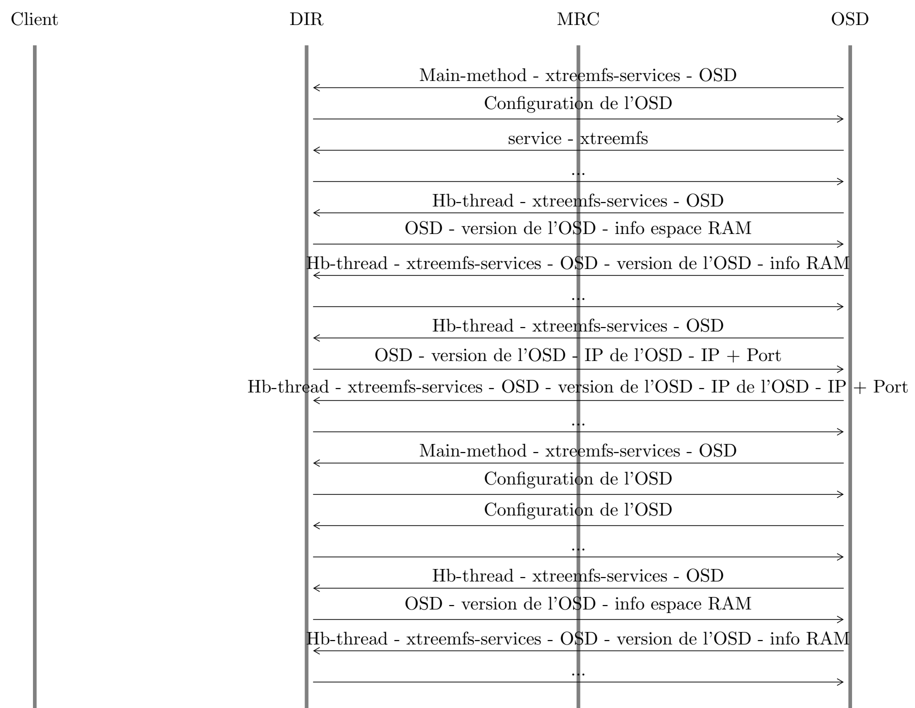

FIGURE 5.5 – Paquets échangés lors de xtreemfs-osd restart

première série d'échanges a lieu entre le client et le MRC, dans laquelle le client indique le nom du fichier au MRC, tandis que le MRC donne l'UUID de l'OSD (ici OSD1) et l'UUID du volume au client. Le client envoie ensuite l'UUID de l'OSD (OSD1) au DIR qui lui renvoie l'adresse IP de cet OSD. Enfin, une fois l'IP de l'OSD connue par le client, ce dernier transmet le contenu du fichier vers l'OSD.

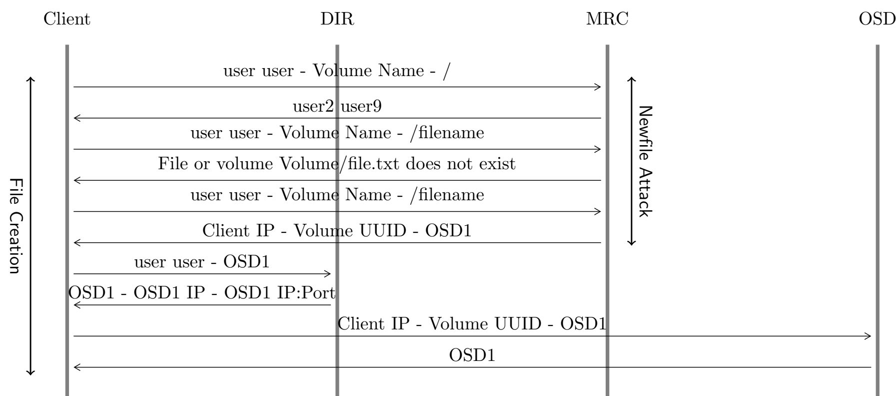

FIGURE 5.6 – Paquets échangés lors de la création d'un fichier

Dans la figure 5.6 on observe lors des différents échanges que les méta-données du fichier sont stockées sur le MRC avant que le fichier ne soit réellement écrit sur l'OSD. Il est donc possible de créer un état incohérent en rejouant les échanges avec le MRC, mais pas ceux avec l'OSD. Les méta-données du fichier seront donc présentes dans le MRC alors que son contenu sera inexistant sur l'OSD.

Les messages entre le client et le MRC ont donc été rejoués. Après ce rejeu, le fichier et ses méta-données sont visibles sur le client. Cependant, il est impossible de lire ou d'écrire sur ce fichier. D'autre part, une tentative de lecture du fichier provoque un arrêt du client. À la suite de cette attaque, le système de fichiers se trouve donc dans un état incohérent.

5.2.2.3 Attaque sur la suppression de fichiers

La troisième attaque avait, comme l'attaque précédente, pour objectif de supprimer les méta-données du fichier sans que le contenu soit supprimé à l'aide de la suppression de fichiers. Lorsqu'un client supprime un fichier, des paquets sont échangés avec le MRC, le DIR puis l'OSD, comme sur la figure 5.7. Une première série d'échanges a lieu entre le client et le MRC, dans laquelle le client indique le nom du fichier au MRC, et le MRC donne l'UUID de l'OSD (OSD1) et du volume au client. Le client envoie ensuite

l'UUID de l'OSD (OSD1) au DIR qui lui répond l'adresse IP de cet OSD. Enfin, le client supprime le fichier de l'OSD.

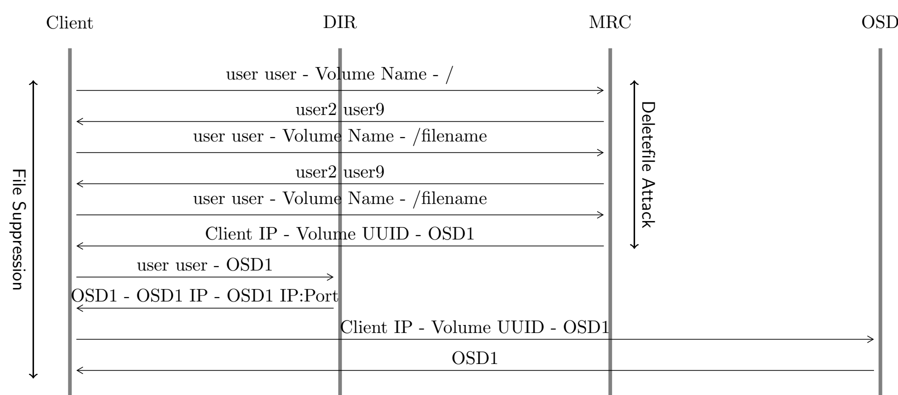

FIGURE 5.7 – Paquets échangés lors de la suppression d'un fichier

On voit dans ces échanges que le fichier est d'abord supprimé sur le MRC, avant que le fichier ne soit réellement supprimé de l'OSD. Le client supprime le fichier de l'OSD une fois que le DIR lui a donné l'adresse de ce dernier. Il est donc possible de créer un état incohérent en rejouant les échanges avec le MRC, mais pas ceux avec l'OSD. On se retrouve alors avec un fichier existant sur l'OSD mais non listé par le MRC. Depuis notre attaquant, nous avons renvoyé les paquets habituellement envoyés par le client au MRC. Nous ne pouvions alors plus voir le fichier avec la commande ls sur notre client. Cependant, les données du fichier n'ont jamais été réellement supprimées et existent toujours sur l'OSD.

5.2.2.4 Attaque sur les permissions d'accès

Dans cette attaque, on souhaite modifier le mode de permission d'accès aux fichiers ou aux répertoires sans avoir le droit d'effectuer cette modification. Lorsqu'un utilisateur souhaite modifier les droits d'accès à un fichier, il va modifier les méta-données du fichier et donc communiquer avec le MRC. Les échanges entre le client et le MRC sont présentés dans la figure 5.8.

Nous avons donc modifié les droits d'accès à un fichier depuis notre attaquant en rejouant les paquets de la figure 5.8. Il est intéressant de noter que l'on peut modifier à la fois les permissions, le nom du volume, le fichier concerné et les utilisateurs sans avoir besoin du mot de passe des utilisateurs ou des permissions. Il est donc possible de modifier les permissions d'accès de n'importe quel utilisateur.

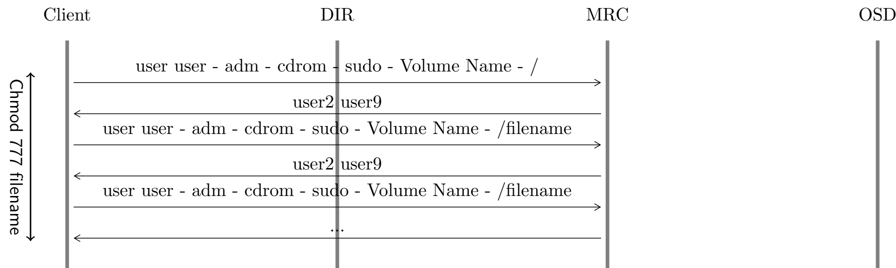

FIGURE 5.8 – Paquets échangés lors de la modification des permissions d'accès d'un fichier

5.2.2.5 Attaque sur le changement de propriétaire

Cette attaque est similaire à l'attaque précédente mais s'intéresse aux propriétaires des fichiers. De même que pour chmod, changer le propriétaire d'un fichier n'implique que des échanges entre le client et le MRC, comme indiqué sur la figure 5.9.

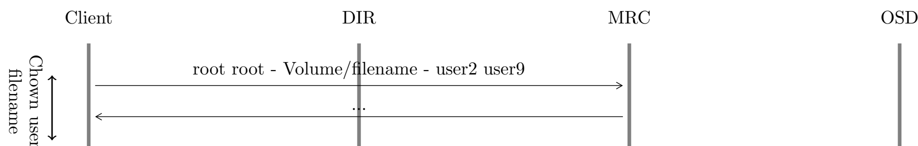

FIGURE 5.9 – Paquets échangés lors de la modification du propriétaire d'un fichier

Nous avons donc pu modifier le propriétaire d'un fichier en rejouant les paquets de la figure 5.9. Cette attaque permet donc d'outrepasser la vérification des droits d'accès à un fichier. Cette fois encore, il est donc possible de changer le nom de l'utilisateur, le nom du volume et du fichier afin d'adapter l'attaque au système visé et aux attentes.

5.2.3 Collecte de données

Pour réaliser la collection de traces et l'exécution des différentes attaques dans le cadre de cette thèse, nous utilisons une architecture constituée de 5 raspberry PI et éventuellement d'une machine pour effectuer certaines attaques. Les raspberry PI hebergent 5 composants du système de fichiers XtreemFS : un MRC, un DIR, deux OSD et un client. Une version non sécurisée de XtreemFS a été préalablement instrumentée pour générer des fichiers de logs contenant à la fois du traffic normal et anormal.

La collecte de données légitimes a été réalisées sur une centaine d'exécutions. Les traces contenaient soit des actions similaires à celles effectuées lors d'attaques soit des actions se rapprochant le plus possible de celles effectuées lors des attaques. Lors de la

production de ces traces correspondant à des exécutions correctes, plusieurs variations ont été effectuées. Ces variations concernent les ordres de lancement ou de terminaison des différents composants d'XtreemFS, le nombre de composants actifs (4 ou 5), le nombre et le type d'actions effectuées, le temps d'exécution ou encore l'ordre dans lequel les actions ont été effectuées.

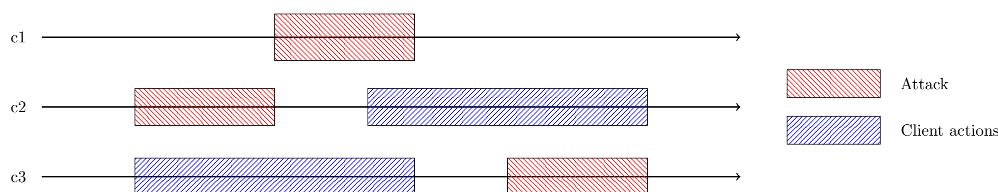

Figure 5.10 – Contextes d'attaques

La collecte de données illégitimes concerne les attaques détaillées dans les parties précédentes. Pour évaluer le système de détection d'intrusion, nous réalisons les attaques selon 3 contextes d'attaques internes (cf figure 5.10) : aucun client n'est actif lors de la réalisation de l'attaque (c1), l'attaque s'effectue avant les actions du client (c2) et l'attaque a lieu après les actions du client (c3). Elles ont aussi été réalisées selon un contexte d'attaque externe où chaque attaque est réalisée depuis une autre adresse IP que celle du client (c4).

Ce jeu de données sera utilisé dans les parties suivantes pour évaluer la solution proposée en terme de passage à l'échelle et de précision du modèle.

5.3 Évaluation des performances

L'approche proposée débute par la construction d'un modèle de comportement d'une application normale (modèle noté `Mod`) composé de deux sous-modèles : un automate (sous-modèle noté `Aut`) et une liste d'invariants (sous-modèle noté `Inv`). Ce modèle est par la suite utilisé lors d'une phase de détection pour évaluer une trace (non apprise). Lors de cette phase, les différents sous-modèles (automates et invariants) sont utilisés pour évaluer la nouvelle trace. Dans cette section, nous nous focalisons sur le temps de construction du modèle et sur les tailles des sous-modèles.

5.3.1 Passage à l'échelle lors de la phase de modélisation

Nous nous intéressons tout d'abord à l'évaluation des deux propositions qui doivent permettre de mieux passer à l'échelle lors de la construction du modèle. Cette évaluation se fait en considérant une application réaliste, relativement riche et complexe en terme de comportements, à savoir XtreemFS (voir section 5.2).

La **première solution proposée** qui est décrite en section 4.2 (*i.e.* sélectionner un sous-ensemble de x séquences) vise à réduire à la fois i) le temps requis pour construire un automate intermédiaire et ii) sa taille. Pour vérifier son efficacité, nous considérons des traces de tailles différentes. Les données correspondant aux constructions indépendantes des modèles intermédiaires sont analysées. En nous basant sur les modèles intermédiaires, nous n'avons pas à considérer les stratégies et les paramètres utilisés ensuite dans la phase de fusion et généralisation (valeur de k par exemple). La figure 5.11 montre la taille de l'automate créé (nombre de transitions) et son temps de construction (secondes) dans trois cas : 1) toutes les séquences du treillis sont considérées ; 2) n heuristiques sont utilisés pour sélectionner n séquences (ici $n = 5$: 1 DIR + 1 MRC + 2 OSD + 1 client) ; et 3) une seule d'entre elles est sélectionnée. Les heuristiques appliquées pour sélectionner des séquences dans le cas 2 sont basées sur les priorités statiques suivantes entre les processus : $p_1 \rhd p_2 \rhd p_3 \rhd p_4 \rhd p_5$, $p_2 \rhd p_3 \rhd p_4 \rhd p_5 \rhd p_1$, $p_3 \rhd p_4 \rhd p_5 \rhd p_1 \rhd p_2$, $p_4 \rhd p_5 \rhd p_1 \rhd p_2 \rhd p_3$, et $p_5 \rhd p_1 \rhd p_2 \rhd p_3 \rhd p_4$. La sélection de n séquences évite la complexité spatiale et temporelle exponentielle d'une approche basée sur le treillis complet (prise en compte de l'ensemble des séquences) .

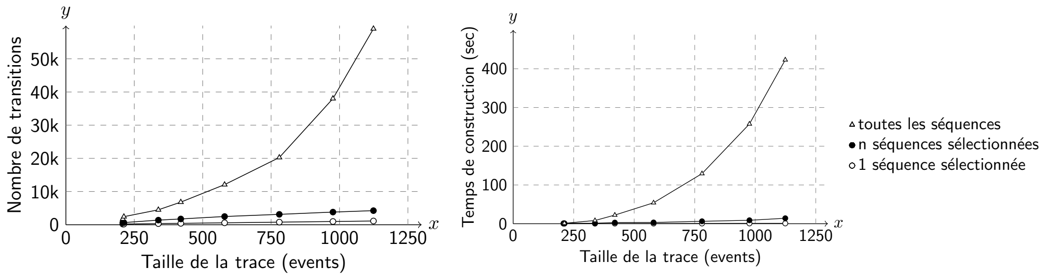

FIGURE 5.11 – Construction d'un automate intermédiaire : complexités spatiale (gauche) et temporelle (droite)

Examinons maintenant l'impact sur la liste des invariants. Le nombre d'invariants reconnus dépend du nombre de séquences sélectionnées. Le tableau 5.1 montre le nombre d'invariants calculés lors de la construction du modèle intermédiaire correspondant, en considérant les mêmes cas et les mêmes tailles de trace que sur la Fig. 5.11. Lorsqu'une seule séquence est sélectionnée ($x = 1$), le modèle est plus restrictif car plus d'invariants sont satisfaits par la seule séquence sélectionnée. Ce résultat est logique puisque la prise en compte de séquences supplémentaires ne peut que conduire à invalider des invariants jusqu'alors considérés comme possibles. Pourtant, la différence reste relativement faible : le taux d'invariants introduits par rapport aux deux autres approches est inférieur à 5 % quelque soit la taille de la trace. Lorsque seules $n = 5$ séquences sont sélectionnées, la détection est aussi précise que lorsque l'on considère l'intégralité du treillis (les lignes

correspondantes dans le tableau 5.1 sont égales). Ce résultat non intuitif et très positif valide le choix d'utiliser un nombre limité de séquences sélectionnées (et des règles de sélection appropriées) pour le calcul des invariants de causalité.

Taille de la trace (events)	212	338	420	581	778	976	1124
1 séquence sélectionnée	3226	4721	6389	12230	16332	17295	17392
n séquences sélectionnées	3118	4643	6231	12082	16149	17084	17108
toutes les séquences	3118	4643	6231	12082	16149	17084	17108

Tableau 5.1 – Taille d'une liste d'invariants intermédiaire

La **deuxième solution proposée** qui est décrite en section 4.3 (*i.e.* exécuter des appels répétés au module *Model Merger & Generalizer* plutôt qu'un seul appel) vise à réduire le temps de calcul lorsque le nombre d'exécutions apprises et/ou leur durée augmente. Dans la figure 5.12, nous comparons les deux approches proposées à l'aide de quatre types de modèles qui correspondent à des processus différents de construction des automates. Les quatre modèles sont nommés `Mod-A-R-1`, `Mod-A-S-1`, `Mod-n-R-1` et `Mod-n-S-1`, où la lettre `R` signifie « Appels répétés » et la lettre `S` signifie « Appel unique (Single) ». La lettre qui apparaît après le mot Mod (pour Modèle) est un `A` pour indiquer que toutes les séquences du treillis sont utilisés (All) ou un `n` si uniquement n séquences sont sélectionnées. Enfin, le dernier chiffre (égal à 1 dans les quatre cas considérés ici) correspond à la valeur du paramètre k utilisé dans l'algorithme k-tail. Pour des nombres différents de traces, nous indiquons le temps requis pour calculer l'ensemble du modèle. La solution proposée (appels répétés) nous permet d'obtenir de meilleurs résultats lorsque le nombre d'exécutions apprises augmente. Les courbes correspondant à un seul appel suivent plus ou moins le nombre d'états de comparaison $\frac{y(y-1)}{2}$ qui sont effectués lors de chaque appel unique. Dans la figure 5.12, chaque modèle intermédiaire est construit soit en utilisant seulement $n = 5$ séquences sélectionnées (haut) soit en utilisant l'ensemble du treillis A (bas). Ainsi, pour les modèles `Mod-n-R-1` et `Mod-n-S-1`, la prise en compte de 80 traces nécessite environ 400 secondes pour un seul appel et seulement 80 secondes en cas d'appels répétés. Lorsque l'on considère l'ensemble du treillis, la différence (entre `Mod-A-R-1` et `Mod-A-S-1`) est encore plus impressionnante. Les appels multiples sont donc beaucoup moins coûteux qu'une fusion et une généralisation globale.

5.3.2 Impact d'un modèle dual sur le temps de détection

Dans cette section, nous cherchons à évaluer l'intérêt d'une utilisation simultanée de différents sous-modèles lors du processus de détection. Nous allons donc évaluer le modèle

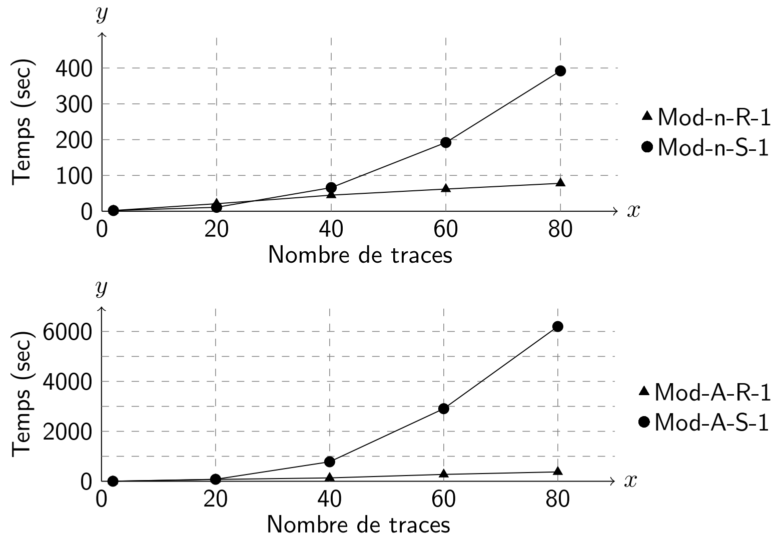

FIGURE 5.12 – Fusion et généralisation : fusion globale versus fusion itérative

complet (`Mod`) et une variante de ce modèle où le couplage entre les deux sous-modèles est moins fort durant la phase de détection.

Dans la solution proposée (`Mod`) qui a été présentée dans la section 3.4, le non respect d'un invariant (détectable en cours d'exploration d'un chemin pour certaines classes d'invariants) provoque un retour arrière et l'évaluation d'un nouveau chemin même si le précédent chemin était à ce stade accepté par l'automate. Dans la variante que nous considérons, un parcours en profondeur à l'aide de l'automate du modèle n'est pas interrompu tant que le préfixe du chemin est reconnu par l'automate. Les invariants ne sont évalués qu'à la fin de l'exploration d'un chemin.

Pour évaluer ces deux versions de l'algorithme, nous considérons trois catégories de traces : les « traces utilisées » correspondent aux traces correctes employées durant le processus de création du modèle ; les « traces non utilisées » correspondent à des traces correctes qui n'ont pas été utilisées pour créer le modèle et les « traces contenant des attaques » qui comme les « traces non utilisées » n'ont pas été utilisées pour créer le modèle mais qui sont incorrectes.

	traces utilisées	traces non utilisées	traces contenant des attaques
Modèle `Mod`	$\leq$ 1 seconde	$\leq$ 1 seconde	$\leq$ 1 seconde
Variante	$\leq$ 1 seconde	$\leq$ 1 seconde - $\geq$ 2 heures	15 minutes - $\geq$ 2 heures

TABLEAU 5.2 – Temps de détection

Le tableau 5.2 présente les temps de détection pour les deux versions de l'algorithme de détection. On observe que les « traces utilisées » sont reconnues en moins de 1 secondes

pour les deux versions de l'algorithme. Cela s'explique par le fait que n'importe quelle séquence qui est compatible avec l'ordre partiel de la trace analysée est nécessairement acceptée par l'automate du modèle et respecte les invariants puisque que les deux sous-modèles ont été construits en tenant compte de ce possible ordre total. Ainsi, dans les deux versions de l'algorithme, la détection ne nécessite de parcourir qu'un seul chemin, le premier choisi. Un constat similaire est parfois valide pour certaines traces correctes mais non utilisées : l'automate généralisé peut très bien accepter le premier chemin évalué. Cependant pour d'autres traces de cet ensemble, le temps de détection peut se révéler particulièrement long dans le cas de la variante alors qu'il n'augmente pas avec le modèle `Mod`. Le temps de détection important s'explique par le fait que l'ensemble des chemins de la trace n'est pas forcément présent dans l'automate généralisé. D'autre part, la différence entre les temps de détection pour les deux versions d'algorithmes est liée au fait que, dans la variante de l'algorithme, l'exploration de chemins n'est pas interrompue dès qu'un préfixe du chemin viole certains invariants. On observe le même phénomène pour les traces contenant des attaques.

Un algorithme de détection vérifiant certains invariants à chaque étape n'améliore pas le temps moyen de détection dans le cas de traces ayant été utilisées pour construire le modèle. Mais, dans le cas des analyses menées avec l'application XtreemFS, il permet pour l'ensemble des traces d'avoir un temps de détection qui reste faible (ce qui n'est pas le cas pour la variante de l'algorithme). L'utilisation combinée des deux sous-modèles présente donc un avantage en terme de temps de détection.

5.4 Évaluation de la qualité du modèle

5.4.1 Capacités de détection

Nous souhaitons résoudre les problèmes de passage à l'échelle sans impacter trop négativement les capacités de détection de l'approche. Pour mesurer l'impact de nos propositions sur l'approche, nous nous proposons de comparer leurs capacités de détection. Le tableau 5.3 montre les résultats de détection pour les 5 attaques et 4 contextes décrits dans la section 5.2.3, en considérant uniquement les stratégies `Mod-A-S-1` et `Mod-n-R-1`. Les sous-modèles automate (`Aut`) et invariants (`Inv`) sont évalués séparément, pour montrer leur complémentarité et valider le choix de conserver les deux. Nous utilisons les notations `Aut` et `Inv` pour marquer la différence avec un modèle dual (notation `Mod`). Même s'il ne peut pas toujours être implémenté, nous indiquons par une coche si l'invariant sur les communications appelé `InvCom` (voir Section 3.4) est capable de détecter l'attaque. Pour chaque stratégie, nous construisons cinq modèles différents en utilisant un sous-ensemble de traces différent lors de chaque phase d'apprentissage.

Dans le tableau, un résultat $d/5$ signifie que d modèles sur cinq ont détecté l'attaque. Les cellules du tableau contenant le symbole « - » sont équivalente à 0/5 : elles indiquent que l'attaque n'a été détectée par aucun des cinq modèles construits en respectant les caractéristiques du type de modèle identifié au début de la ligne du tableau.

Attaque	*NewFile*				*DeleteFile*				*OsdChange*				*Chmod*				*Chown*			
Contexte	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4
`Aut-A-S-1`	-	2/5	-	5/5	-	3/5	1/5	5/5	2/5	5/5	4/5	5/5	-	1/5	-	5/5	-	1/5	-	-
`Aut-n-R-1`	-	-	-	5/5	-	1/5	-	5/5	4/5	4/5	4/5	5/5	-	-	-	5/5	-	-	-	-
`Inv-A-S-1`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`Inv-n-R-1`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`InvCom`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	-

Tableau 5.3 – Detection of attacks in various contexts using different sub-models

Premier constat, pour une même application, une même attaque et un même modèle, la qualité de la détection varie parfois fortement en fonction du contexte. Ceci est vrai pour les deux sous-modèles et s'explique notamment par le fait que l'activité correcte qui précède ou suit une attaque influence la progression dans les automates (celui associé au sous-modèle `Aut` et ceux associés à chaque invariant du sous-modèle `Inv`). Ce constat justifie donc la prise en compte des contextes dans notre méthodologie d'évaluation, ce qui n'est pas une pratique répandue.

Les résultats montrent que les invariants sont utiles pour détecter les attaques dans les contextes c1 et c2, tandis que les automates fonctionnent mieux dans le contexte c4. Les solutions proposées pour améliorer le passage à l'échelle n'ont aucun impact sur les capacités de détection des invariants, mais elles dégradent légèrement celles des automates. Cependant, notre modèle est composé des deux sous-modèles (automate et invariants). Ainsi, une alerte est déclenchée lorsque la trace analysée est rejetée par le sous-modèle `Inv` ou par le sous-modèle `Aut`. La détection basée sur des invariants masque les erreurs possibles du sous-modèle automates dans 8 des 20 cas. Dans les 12 cas restants, les résultats fournis par les sous-modèles `Aut-A-S-1` et `Aut-n-R-1` sont les mêmes, sauf dans 2 cas (à savoir, l'attaque *DeleteFile* dans le contexte c3 et l'attaque *Chown* dans le contexte c2). Dans ces deux cas particuliers, la qualité de la détection est assez similaire : les scores sont de 0/5 et 1/5. Ainsi, en termes de faux négatifs, les solutions proposées pour pouvoir passer à l'échelle n'ont quasiment aucun impact sur l'approche. Les différences sont faibles ou masquées par le fait que le modèle est composé de deux sous-modèles.

Le tableau 5.4 montre les résultats de 6 stratégies distinctes pour construire le sous-modèle d'automate. Chaque pourcentage indique la proportion d'attaques détectées, sa-

#	Modèle	*NewFile*	*DeleteFile*	*OsdChange*	*Chmod*	*Chown*
1	`Aut-A-S-5`	100%	100%	100%	100%	100%
2	`Aut-A-S-1`	35%	45%	80%	30%	5%
3	`Aut-A-R-1`	30%	35%	50%	30%	5%
4	`Aut-n-S-5`	100%	100%	100%	100%	100%
5	`Aut-n-S-1`	25%	25%	85%	25%	0%
6	`Aut-n-R-1`	25%	30%	85%	25%	0%

TABLEAU 5.4 – Détection d'attaques avec différentes stratégies de construction d'automate

chant que 4 contextes et 5 modèles ont été utilisés pour tester chaque stratégie. Comme nous nous concentrons uniquement sur le sous-modèle automate, les pourcentages sont bien inférieurs à ceux correspondant au modèle dual (rappelons que le sous-modèle invariant peut détecter jusqu'à la moitié des quatre premières attaques). Lorsqu'aucune généralisation n'est effectuée (*ie* pour une valeur élevée de k dans l'algorithme *k-tail*, en l'occurrence $k = 5$), aucun faux négatif ne se produit (voir lignes 1 et 4 dans le tableau 5.4). Cependant, tous les comportements corrects qui n'ont pas été appris vont déclencher des alertes (faux positifs). La stratégie ne passe cependant pas à l'échelle et donc l'adopter peut se révéler coûteux (pour un grand nombre de traces). Aux lignes 2 et 3 (resp. 5 et 6), un modèle intermédiaire est construit en utilisant le treillis entier (resp. en sélectionnant n séquences). Les pourcentages de détection présentés dans ces quatre lignes indiquent des différences (un écart généralement de l'ordre de 5 à 10 points). La technique de passage a l'échelle a donc un petit impact négatif sur la qualité de détection via le modèle `Aut` mais cet impact est limité par l'utilisation du deuxième sous-modèle (invariants).

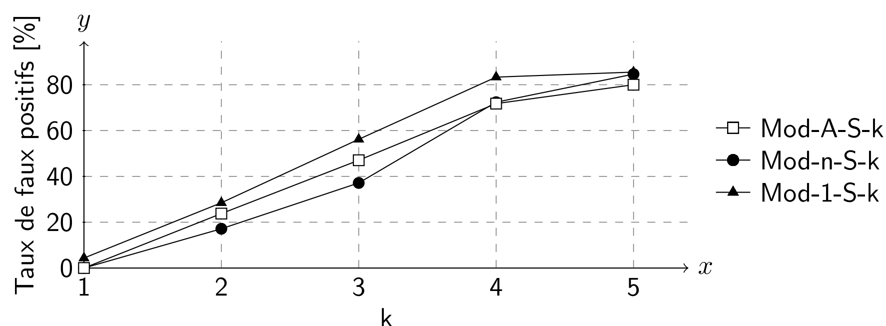

FIGURE 5.13 – Taux de faux positifs pour différentes valeurs de k et x

La précision du modèle doit être évaluée en termes de faux négatifs mais aussi de faux positifs. Nous nous concentrons ici sur l'impact de la sélection de x séquences sur le taux de faux positifs. La figure 5.13 montre un résultat bien connu : plus la valeur

de k dans l'algorithme *k-tail* est petite, plus l'automate résultant de cette généralisation accepte de séquences. Cela souligne l'importance de la phase de généralisation. Pour une valeur donnée de k, on observe également des différences en fonction du nombre x de séquences sélectionnées. Lorsque x est trop petit (égal à 1), moins de comportements sont acceptés. Mais les meilleurs résultats sont obtenus lorsque n séquences sont sélectionnées et non lorsque le treillis entier est utilisé. Ce bon résultat est à nouveau en faveur de notre solution proposée. Cela peut s'expliquer par le fait que, lorsque $x = n$, les automates fournis en entrée pour la phase de fusion et de généralisation ne sont ni trop simples (les n chemins sélectionnés peuvent correspondre à beaucoup plus de chemins conservés) ni trop complexes (l'ensemble du treillis). Tout en conservant une multitude de comportements, ces automates ont souvent moins de nœuds et surtout moins de transitions. En conséquence, l'algorithme *k-tail* peut fusionner plus d'états et générer un automate final acceptant plus de comportements.

D'autres expériences (non rapportées dans les figures de cette section pour des questions de lisibilité) montrent l'intérêt de définir des règles de sélection qui identifient des séquences qui ne sont pas proches les unes des autres. Par exemple, si seulement deux séquences sont sélectionnées et que la première règle de sélection est $p_1 \rhd p_2 \rhd p_3 \rhd p_4 \rhd p_5$, un bon choix pour la deuxième règle est $p_5 \rhd p_4 \rhd p_3 \rhd p_2 \rhd p_1$ et non $p_1 \rhd p_2 \rhd p_3 \rhd p_5 \rhd p_4$. Dans le premier cas, les résultats de détection se rapprocheront de ceux d'une sélection de n chemins. Dans le second cas, les résultats se rapprocheront d'une sélection de 1 chemin. Ainsi, les résultats liés à l'approche sont impactés à la fois par le nombre de chemins choisis pour construire le modèle mais aussi par les heuristiques utilisées pour sélectionner ces chemins.

5.4.2 Complétude du modèle

Pour évaluer la complétude du modèle, nous devons disposer d'une application distribuée avec un nombre fini et dénombrable de comportements. Pour cela, nous considérons le cas d'étude exposé dans la section 5.1 qui est composé de deux scénarios. L'expérimentation considérée dans cette partie a pour but d'évaluer le nombre de traces nécessaires pour construire un modèle de comportement acceptant l'ensemble des comportements possibles.

Ici nous considérons trois cas : 1) l'application n'accepte que le comportement exposé dans le scénario n° 1 (deux clients effectuent un achat auprès d'un serveur), 2) l'application accepte uniquement le scénario n° 2 (le premier client effectue un achat auprès du serveur et le second échoue dans sa recherche) et 3) l'application accepte à la fois les scénarios n° 1 et n° 2 (le premier client effectue un achat auprès du serveur et le second client peut soit effectuer un achat soit échouer dans la recherche d'un produit).

L'expérimentation s'effectue en apprenant plusieurs comportements à partir d'un nombre variable x de traces. Ici x est compris entre 1 et le nombre maximum de traces dénoté x_{max} (par exemple, $x_{max} = 70$ pour le cas n° 1). Dans cette expérience, nous avons construit (pour chaque valeur de x) 10 ensembles de x traces choisies aléatoirement Ces ensembles permettent de construire des modèles selon l'approche `Mod-n-R-1` qui sont ensuite évalués sur l'ensemble des traces possibles.

Taux d'acceptation (par rapport à 10 modèles construits) de l'ensemble de toutes les traces possibles		$\geq$ 1/10 x_n/x_{max}	$\geq$ 9/10 x_s/x_{max}
Automates	Cas n° 1	12/70	17/70
	Cas n° 2	6/15	8/15
	Cas n° 3	14/85	20/85
Invariants	Cas n° 1	3/70	6/70
	Cas n° 2	3/15	5/15
	Cas n° 3	6/85	10/85

TABLEAU 5.5 – Nombre nécessaire de traces x_n et nombre suffisant de traces x_s

Le tableau 5.5 présente les résultats de l'expérimentation. Dans ce tableau, les deux colonnes les plus à droite identifient un nombre nécessaire de traces et un nombre suffisant de traces qui doivent être utilisées pour construire un modèle de comportement. Ce nombre nécessaire x_n (le plus petit) et ce nombre suffisant x_s (le plus grand) sont définis de la façon suivante. x_n est la plus petite valeur de x pour laquelle au moins un modèle parmi les dix générés a permis d'accepter l'ensemble des comportements (appris ou pas). x_s est la plus petite valeur de x pour laquelle quasiment tous les modèles ont permis de détecter l'ensemble des comportements possibles. L'adverbe quasiment est utilisé car nous avons fixé le seuil à neuf modèles sur dix pour considérer que la condition est remplie. Ces deux valeurs permettent d'avoir une fourchette permettant d'estimer la qualité de l'apprentissage en fonction du nombre de traces apprises.

Par exemple, pour le cas n° 1, à la première ligne, on dispose de 70 traces. Parmi ces 70 traces, on construit 10 groupes de x traces pour différentes valeurs de x. Les tests effectués permettent de constater qu'il faut utiliser des groupes de 12 traces pour qu'un des dix modèles construits accepte (via le sous-modèle `Aut`) l'ensemble des traces. Il faut utiliser des groupes de 17 traces pour que 9 modèles sur dix construits acceptent (toujours via le sous-modèle `Aut`) l'ensemble des comportements.

A partir des résultats de ce tableau, on peut effectuer plusieurs observations. Premièrement, il est possible de construire un modèle de comportement acceptant l'ensemble des traces d'exécutions possibles avec un sous-ensemble de ces traces. L'écart entre la valeur x_n et la valeur x_s est plutôt assez faible : l'amélioration de la qualité du modèle est

donc assez rapide. On remarque également que le sous-modèle que constitue la liste d'invariants nécessite en général moins de traces que les automates pour être complet pour les différents cas. Enfin dans le cas n° 3, comportant à la fois des comportements des cas n° 1 et n° 2, moins de traces sont nécessaires pour construire un modèle de comportement complet. En effet, si on effectue une addition entre le nombre de traces « suffisantes » pour accepter l'ensemble des comportements des cas n° 1 et n° 2 on se retrouve avec un ensemble de 25 traces (respectivement 17 traces pour le cas n° 1 et 8 traces pour le cas n° 2). Cependant, 20 traces permettent de faire en sorte que la quasi totalité des modèles accepte l'ensemble des comportements dans le cas n° 3. Cela s'explique par le fait qu'une partie des types d'événements présents dans le scénarios n° 2 sont déjà inclus dans scénario n° 1.

5.4.3 Acceptation de traces de tailles variables

Les traces utilisées lors de la phase d'apprentissage peuvent correspondre à des exécutions ayant des durées plus ou moins longues. En utilisant des traces courtes (d'une durée de 2 minutes à 5 minutes), nous avons montré qu'il était possible de produire le modèle de comportement d'une application distribuée pour la détection d'intrusion. La proposition effectuée pour passer à l'échelle permet de construire un modèle (`Mod-n-R-k`) en temps linéaire par rapport au nombre de traces sans avoir un impact significatif sur les capacités de détection. A présent, nous considérons l'étude d'exécutions plus longues. Alors que la phase de construction repose sur l'utilisation de traces courtes (au moins 32 traces), les traces correspondant à des durées d'exécutions plus importantes (5 heures) sont acceptées par le modèle lors de la phase de détection. Il semble donc intéressant d'avoir une stratégie d'apprentissage basée principalement sur des traces correspondant à des exécutions courtes.

Le tableau 5.6 montre des valeurs moyennes permettant de comparer les performances de notre approche pour des traces d'exécution courtes (5 minutes) et longues (5 heures) en entrée et ceci pour chaque phase (construction et fusion/généralisation). En ce qui concerne la phase de construction, les résultats dans la première colonne correspondent à peu près à ceux décrits dans la figure 5.11 et le tableau 5.1. La trace contient plus d'événements lorsque l'exécution dure 5 heures. Ainsi, le temps de construction d'un modèle intermédiaire et le nombre de transitions sont tous deux plus importants. Notez que les ratios pour les résultats en phase de construction sont d'environ 35, à l'exception du nombre d'invariants, qui est assez similaire avec des traces courtes ou longues. Cela est dû au fait que l'activité réalisée était similaire dans les deux exécutions : les mêmes types d'événements ont été observés.

Phase	construction			fusion & généralisation		
Durée d'exécution	5 min	5 hr	*ratio*	5 min	5 hr	*ratio*
Taille de la trace (événements)	754	26178	34.7	754	26178	34.7
Temps de calcul (seconds)	5	178	35.6	8	386	48.2
Nombre de transitions	2502	87266	34.9	386	247	0.6
Taille de la liste d'invariants	16643	15239	0.9	2125	2058	0.9
Taille de la liste d'invariants réduite	-	-	-	1190	986	0.8

Tableau 5.6 – Performances pour chaque phases

Concernant la phase de fusion & généralisation, le temps nécessaire pour fusionner et généraliser les traces courtes correspond approximativement à celui indiqué sur la figure 5.12. Bien sûr, il faut beaucoup plus de temps pour fusionner et généraliser les modèles intermédiaires obtenus après une longue exécution (en raison du plus grand nombre de nœuds et de transitions). Pourtant, nous observons que les tailles de l'automate de résultat et la liste invariante sont comparables dans les deux cas. Cela est également dû au fait que, lors de nos tests, les exécutions plus longues ne révèlent pas de nouveaux comportements. La dernière ligne du tableau 5.6 indique la taille réduite de la liste invariante après une optimisation simple (basée sur les relations de transitivité) qui n'affecte pas la précision du sous-modèle `Inv-n-R-k` : par exemple, lorsqu'une liste contient les invariants $a \rightarrow b$, $a \rightarrow c$, and $b \rightarrow c$, nous n'avons qu'à garder $a \rightarrow b$ et $b \rightarrow c$.

5.5 Conclusion

Ce chapitre a présenté une évaluation de l'approche de modélisation proposée pour la détection d'intrusion. Pour évaluer l'approche, deux cas d'études sont d'abord introduits. Le premier cas d'étude concerne une application distribuée dont on connaît l'ensemble de comportements possibles. Le second traite d'une application distribuée plus complexe. Des attaques contre cette dernière ont par ailleurs été exposée dans un objectif d'évaluation. A la suite de ces deux cas d'étude, des évaluations de l'approche ont été réalisées. Une première évaluation a consisté à évaluer l'approche en terme de performance (temps nécessaire à la construction du modèle et taille du modèle) et à valider le fait que nos deux propositions permettent d'améliorer significativement ces performances. Cette évaluation a permis de confirmer que l'approche initiale souffrait effectivement de problèmes de passage à l'échelle. Les deux propositions introduites dans le chapitre 4 permettent d'adresser ce problème. Une seconde évaluation avait pour objectif d'évaluer la qualité du modèle. L'évaluation a permis de montrer que l'approche

était utile pour détecter des attaques contre l'intégrité d'un système distribué. L'utilisation de plusieurs sous-modèles complémentaires permettent de conserver des résultats de détection similaires que les deux propositions pour passer à l'échelle soient employées ou non. Par ailleurs, on a pu observer qu'il était possible de construire un modèle de comportement acceptant l'ensemble des comportements de l'application à l'aide d'un nombre réduit de traces et qu'il était possible d'utiliser des traces d'exécution courtes lors de la phase d'apprentissage pour obtenir un modèle capable d'accepter des traces plus longues.

Chapitre 6

Propositions de modèles complémentaires

Ce chapitre considère des modèles de comportement alternatifs ou additionnels aux deux modèles jusqu'alors étudiés. Plus précisément, deux types de modèles possibles sont envisagés. Dans un premier temps, nous étudions des ajouts possibles aux invariants existants. Ensuite, nous explorons un modèle basé sur l'utilisation de n-grammes (en anglais, ngrams).

6.1 Invariants

Les invariants utilisés dans le contexte de cette thèse et de nombreux travaux de l'état de l'art correspondent aux invariants les plus fréquents de l'article de Dwyer et al. [DAC99]. Cependant, les approches de l'état de l'art qui utilisent ces invariants pour inférer des automates [THH+16, LFY+10, BBEK14] n'explorent pas la possibilité d'utiliser des « scopes » (fenêtres temporelles). D'autre part, il est possible d'utiliser lors du processus d'inférence d'un modèle d'autres types de propriétés portant par exemple sur le nombre d'occurrence de chaque type d'événements [LFY+10]. Dans cette section nous explorons deux propositions : la première concerne les invariants d'occurrence et la seconde traite des fenêtres temporelles.

6.1.1 Invariants d'occurrence

Dans l'article [LFY+10], les auteurs utilisent des propriétés d'occurrence en plus des invariants de causalité pour construire un modèle de comportement d'un programme. Dans ce travail, les propriétés d'occurrences sont utilisées pour déterminer le type des états qui peuvent soit être des « switch states » (exécution séquentielle) soit des « split/fork states » (exécution concurrente). L'utilisation de ce type de propriété comme modèle

complémentaire à la liste d'invariants de causalité et à l'automate semble donc intéressante.

6.1.1.1 Proposition

Pour utiliser ce type de propriétés comme modèle complémentaire à l'automate et à la liste d'invariants de causalité, nous proposons de générer une liste d'invariants d'occurrence pour chaque trace d'exécution. Les propriétés sont évaluées en considérant l'exécution dans son intégralité (fenêtre temporelle globale). Les listes obtenues pour chaque trace apprise sont ensuite fusionnées pour obtenir une unique liste d'invariants d'occurrence qui correspond à l'ensemble des exécutions apprises. Cette liste sera utilisée, tout comme la liste d'invariants de causalité, comme modèle complémentaire lors de la phase de détection. Pour chaque couple de type d'événements a et b, les trois types de propriétés suivants sont considérés :

— le nombre d'occurrences d'événements de type a est égal au nombre d'occurrences d'événements de type b, noté $\mathrm{Occur(a)} = \mathrm{Occur(b)}$

— le nombre d'occurrences d'événements de type a est supérieur au nombre d'occurrences d'événements de type b, noté $\mathrm{Occur(a)} > \mathrm{Occur(b)}$

— le nombre d'occurrences d'événements de type a est supérieur ou égal au nombre d'occurrences d'événements de type b, noté $\mathrm{Occur(a)} \geq \mathrm{Occur(b)}$

Parmi les invariants proposés, ne figurent pas les opérateurs $\leq$ et $<$ pour des raisons de simplicité et de redondance. Ces deux opérateurs peuvent être exprimés à l'aide des opérateurs utilisés. Par exemple, l'invariant $\mathrm{Occur(a)} \geq \mathrm{Occur(b)}$ correspond à l'invariant $\mathrm{Occur(b)} \leq \mathrm{Occur(a)}$. Ces invariants proposés sont complémentaires aux invariants de causalité car ils n'expriment pas les mêmes contraintes. Par exemple, l'invariant a $\rightarrow$ b permet d'accepter les séquences : <a, a, a, a, b>, <a, a, a, b, b, b>, <a, b, a, b, a, b, a, b ,a, b> et <a, b, b, b, b>. Avec un invariant d'occurrence comme $\mathrm{Occur(a)} \geq \mathrm{Occur(b)}$, les trois premières séquences décrites ci-dessus seront acceptées alors que la dernière sera rejetée.

6.1.1.2 Inférence d'invariants d'occurrence à partir d'une trace d'exécution

Le nombre d'occurrence d'un type d'événement correspond au nombre de fois que ce type d'événement apparaît dans une trace d'exécution. Pour inférer un invariant d'occurrence, on a besoin d'obtenir le nombre d'apparitions des deux types d'événements concernés. L'inférence se réalise en deux phases.

La première phase consiste à comptabiliser pour chaque type d'événement le nombre de fois où il apparaît. Pour cela, on peut soit analyser directement la trace, soit effectuer un comptage durant la construction des autres modèles au moment de l'exploration de la première séquence complète d'événements. Dans notre cas, le calcul a été fait lors d'une analyse séparée de la trace. Pour chaque événement de la trace, on met à jour le nombre d'occurrence des différents types d'événements dans une hashmap : *nbOccur<EventType*, *Int>*. Dans cette hashmap, *EventType* correspond aux types d'événements qui vont être dénombrés et *Int* au nombre d'occurrence de ces types d'événements.

Une seconde phase vise à comparer les nombres d'occurrence des événements pour les classer en deux catégories d'invariants définis précédemment : Occur (a) > Occur (b) et Occur (a) = Occur (b). Le troisième type d'invariants ne peut être inféré lors de la phase de fusion des invariants.

Algorithm 11 calcul des invariants d'occurrence

```
 1: function inferOccur
 2:     for i ← 0 to listEventType.size() do
 3:         e1 ← listEventType.get(i)
 4:         for j = i + 1 to listEventType.size() do
 5:             e2 ← listEventType.get(j)
 6:             if ncOccur.get(e1) > nbOccur.get(e2) then
 7:                 suplist.add(e1, e2)
 8:             else if ncOccur.get(e1) == nbOccur.get(e2) then
 9:                 eqlist.add(e1, e2)
10:             else
11:                 suplist.add(e2, e1)
12:             end if
13:         end for
14:     end for
15: end function
```

L'algorithme 11 détaille la fonction qui permet de calculer les invariants d'occurrence. Dans cet algorithme, la liste est explorée et toutes les occurrences des différents types d'événements sont comparées deux à deux. Le résultat de cette comparaison (effectuées dans les lignes 6, 8 et 10) permet de créer l'invariant qui correspond à la relation entre les occurrences. Les invariants sont ensuite stockés dans une liste qui contiendra l'ensemble des invariants de l'exécution. L'algorithme 11 est similaire à l'algorithme 5 et présente des complexités spatiales et temporelles semblables.

6.1.1.3 Fusion des listes d'invariants d'occurrence

Après avoir calculé les listes d'invariants pour toutes les exécutions, il est possible de fusionner les listes d'invariants deux à deux tout comme dans la section 3.3.2. L'objectif de cette fusion est d'obtenir une liste d'invariants d'occurrence qui reste vraie pour l'ensemble des exécutions.

Pour fusionner les listes d'invariants, on va comparer les invariants des deux exécutions. Les invariants qui contiennent le même couple d'opérandes sont comparés en priorité. Si la comparaison n'est pas possible (il manque un ou plusieurs types d'événements dans une des deux exécutions), on va tenir compte du ou des types d'événements manquants. Lors de l'étape de fusion, un nouveau type d'invariant peut apparaître : Occur(a) $\geq$ Occur(b). La fusion va appliquer les règles ci-dessous :

Définition 18 $\forall$ *a,* $\forall$ *b,* $Occur(a) > Occur(b) \in I_{\alpha\cap\beta}$ *si :*

- $Occur(a) > Occur(b) \in I_\alpha \wedge Occur(a) > Occur(b) \in I_\beta$
- $Occur(a) > Occur(b) \in I_\alpha \wedge b \notin E_\beta$

Définition 19 $\forall$ *a,* $\forall$ *b,* $Occur(a) \geq Occur(b) \in I_{\alpha\cap\beta}$ *si :*

- $Occur(a) \geq Occur(b) \in I_\alpha \wedge Occur(a) > Occur(b) \in I_\beta$
- $Occur(a) \geq Occur(b) \in I_\alpha \wedge Occur(a) = Occur(b) \in I_\beta$
- $Occur(a) > Occur(b) \in I_\alpha \wedge Occur(a) = Occur(b) \in I_\beta$
- $Occur(a) \geq Occur(b) \in I_\alpha \wedge b \notin E_\beta$
- $Occur(a) = Occur(b) \in I_\alpha \wedge b \notin E_\beta$

Définition 20 $\forall$ *a,* $\forall$ *b,* $Occur(a) = Occur(b) \in I_{\alpha\cap\beta}$ *si :*

- $Occur(a) = Occur(b) \in I_\alpha \wedge Occur(a) = Occur(b) \in I_\beta$
- $Occur(a) = Occur(b) \in I_\alpha \wedge a, b \notin E_\beta$

Dans notre proposition, une phase de réduction n'a pas été envisagée contrairement aux invariants de causalité. Elle pourrait cependant être prévue dans un travail futur. La phase de réduction pourrait en effet s'appuyer comme dans la section 3.3.2 sur la transitivité des relations d'ordre $>$ et $\geq$.

6.1.1.4 Détection

Contrairement à certains types d'invariants de causalité, les invariants d'occurrence sur un scope global ne peuvent être évalués durant l'exploration de la trace. L'évaluation de ce type d'invariant peut se faire à l'aide des fichiers de logs : le nombre d'occurrences des différents types d'événements reste le même quelque soit l'ordre dans lequel les événements se produisent. L'évaluation peut aussi être faite à la fin de l'évaluation de la trace quand un chemin est considéré comme correct par les autres modèles. Dans les deux cas, le processus consiste à établir une hashmap nbOccur<EventType, Int> qui sert à dénombrer le nombre d'occurrences des différents types d'événements. La hashmap sera ensuite utilisée, dans une seconde phase, pour générer une liste d'invariants d'occurrence propre à la trace évaluée. cette liste sera comparée à la liste d'invariants d'occurrence du modèle selon les règles suivantes :

Définition 21 *Eval*(I_δ, I_{global}) = *faux si* $\forall$ *a*, $\forall$ *b* :

- $Occur(a) > Occur(b) \in I_{global} \wedge Occur(b) > Occur(a) \in I_\delta$
- $Occur(a) > Occur(b) \in I_{global} \wedge Occur(a) = Occur(b) \in I_\delta$
- $Occur(a) > Occur(b) \in I_{global} \wedge a \notin E_\delta \wedge b \in E_\delta$
- $Occur(a) = Occur(b) \in I_{global} \wedge a \notin E_\delta \wedge b \in E_\delta$
- $Occur(a) = Occur(b) \in I_{global} \wedge a \in E_\delta \wedge b \notin E_\delta$
- $Occur(a) \geq Occur(b) \in I_{global} \wedge Occur(b) > Occur(a) \in I_\delta$
- $Occur(a) \geq Occur(b) \in I_{global} \wedge a \notin E_\delta \wedge b \in E_\delta$

6.1.1.5 Évaluation

Nous allons maintenant évaluer l'intérêt d'utiliser des invariants d'occurrence. Dans cette section, une évaluation de cette proposition est effectuée à la fois en terme de passage à l'échelle mais aussi en terme de capacité de détection.

La première évaluation effectuée a pour objectif de mesurer le coût de cette proposition qui va venir s'ajouter au coût de l'approche de modélisation présentée dans les chapitres 3 et 4. Pour cela, les temps des phases de construction et de fusion & généralisation spécifiques à cette proposition ainsi que la taille des listes d'invariants d'occurrence générées ont été mesurés. La méthode de génération se basant sur les traces et l'étape de fusion étant complètement décorrélée des fusions concernant les autres types de modèle, il n'y a donc pas besoin de faire une comparaison directe avec les autres modèles. Pour la suite, cette approche sera comparée avec l'approche incluant nos propositions pour passer à l'échelle (`Mod-n-R-k`).

Phase	construction			fusion & généralisation		
Durée d'exécution	5 min	5 hr	*ratio*	5 min		*ratio*
Taille de la trace (événements)	754	26178	34.71	-	-	-
Nombre de traces	1	1	-	20	60	3
Coût en temps (secondes)	0.48	12.458	25.95	4	10	2.5
Taille de la liste d'invariants d'occurrence	1964	2054	1.04	60	96	1.6

Tableau 6.1 – Performance liée à la proposition des invariants d'occurrence

Le tableau 6.1 présente les résultats de performances (valeurs moyennes) liés à l'utilisation d'invariants d'occurrence. Sa structure est similaire à celle du tableau 5.6. Les différences notables avec l'approche traditionnelle apparaissent dans ce tableau au travers du coût en temps et de la taille de la liste des invariants d'occurrence.

Par rapport à la phase de construction, on remarque que les coûts en temps pour les différentes traces sont relativement faibles par rapport à la durée de l'exécution qui a produit la trace. L'impact semble donc peu important sur l'approche de modélisation. D'autre part, le ratio de ce coût en temps est inférieur au ratio de la taille de la trace. L'approche semble donc passer à l'échelle. Concernant les tailles des listes d'invariants d'occurrence, les résultats figurant dans le tableau montrent que, peu importe la durée de l'exécution, la taille de ces listes reste assez stable (ratio de 1). Une observation similaire avait été faite pour les invariants de causalité avec un nombre d'invariants plus élevé (voir le tableau 5.6).

En ce qui concerne la phase de fusion et généralisation, le ratio entre les temps est similaire au ratio lié au nombre de traces. D'autre part, les temps nécessaires pour fusionner les listes ont l'air peu importants. L'approche de fusion des listes d'invariants d'occurrence passe donc à l'échelle. Une fois fusionnées, les différentes listes sont de taille réduite comparée aux listes d'invariants calculées lors de la phase de construction. Notons qu'il existe une différence notable entre les tailles de listes obtenues en utilisant respectivement 60 traces et 20 traces lors de la phase de construction. Cette différence peut avoir un impact sur les capacités de détection lorsque ces deux listes seront exploitées.

La deuxième évaluation a pour objectif de mesurer l'impact de cette proposition sur les capacités de détection. Cette évaluation s'effectue en deux temps. Dans un premier temps, une évaluation en terme de faux positifs est proposée. Dans un second temps, une étude sur la complémentarité des modèles en terme de détection d'attaque est effectuée.

Pour mesurer l'impact d'une liste d'invariants d'occurrence sur le taux de faux positifs, une série de plusieurs modèles ont été construits à partir de x traces (x étant compris entre 2 et 60). Ces modèles ont ensuite été utilisés dans le cadre de la phase de détection pour évaluer des traces d'exécutions correctes.

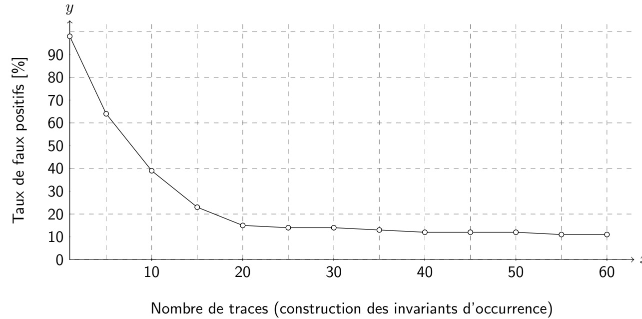

FIGURE 6.1 – Taux de faux positifs du modèle d'invariants d'occurrence

La figure 6.1 présente les taux de faux positifs moyens obtenus pour les différents modèles. Plus le nombre de traces utilisées pour construire cette liste est grand, plus le taux de faux positif diminue. Ce taux se stabilise entre 20% et 10% si plus de 20 traces ont été utilisées pour construire la liste d'invariants d'occurrence. On avait noté dans l'expérimentation précédente une différence entre les tailles de liste d'invariants pour 20 traces et 60 traces. Cette différence correspond dans cette figure à un passage d'un taux de faux positifs de 15% pour 20 traces à 11% pour 60 traces.

Un autre aspect de l'évaluation des capacités de détection du modèle concerne la détection d'attaques. Pour connaître l'intérêt des invariants d'occurrence sur l'approche, nous mesurons leurs capacités de détection. Pour cela, nous effectuons une expérimentation similaire à celle exposée dans la section 5.4.1 en considérant cette fois-ci la liste d'invariants d'occurrence comme modèle additionnel aux autres types de modèles.

Attaque	*NewFile*				*DeleteFile*				*OsdChange*				*Chmod*				*Chown*			
Contexte	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4	c1	c2	c3	c4
`Aut-n-R-1`	-	-	-	5/5	-	1/5	-	5/5	4/5	4/5	4/5	5/5	-	-	-	5/5	-	-	-	-
`Inv-n-R-1`	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	-	-	-	-
`InvOccur`	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	-	-	5/5	5/5	1/5
`InvCom`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-	-	-

TABLEAU 6.2 – Complémentarité dans la détection entre les invariants d'occurrence et les autres sous-modèles

Le tableau 6.2 a une structure similaire au tableau 5.3. Il présente les résultats de détection pour les 5 attaques selon 4 contextes décrits dans la section 5.2.3 pour les différents sous-modèles générés. Dans ce tableau, nous pouvons observer que le nouveau sous-modèle fondé sur les invariants d'occurrence et nommé `InvOccur` est particulièrement efficace pour détecter les attaques dans les contextes c2 et c3 et presque inutile

pour les autres contextes. Une observation similaire a été effectuée dans la section 5.4.1 pour les modèles automate et invariants de causalité sur des contextes d'attaques différents. Dans ce tableau, l'utilisation de plusieurs types de sous-modèles complémentaires permet de réduire le taux de faux négatifs. D'autre part, on remarque que les invariants d'occurrence ont été capables de détecter le dernier type d'attaque (Chown) alors que les autres modèles n'en ont pas été capables.

6.1.2 Introduction de scopes pour les invariants

La proposition briévement décrite dans cette section reprend des principes détaillés dans l'article de Dwyer et al. [DAC99]. Dans cet article, des portées (scope) qui définissent des fenêtres temporelles sont proposées pour exprimer des propriétés. Parmi les articles de l'état de l'art cherchant à inférer un modèle de comportement d'une application, certains utilisent les propriétés décrites dans cet article pour l'inférence de leur modèle [THH+16, LFY+10, BBEK14]. Mais ces articles considèrent la portée « Global » pour inférer leurs invariants : c'est à chaque fois l'intégralité de l'exécution qui est considérée pour évaluer si une propriété est satisfaite ou pas.

6.1.2.1 Proposition

Nous souhaitons inférer des invariants plus complexes en utilisant la notion de portée (scope) d'un invariant. Pour cela, nous nous focalisons sur trois classes d'invariants décrites dans la section 3.2.3, à savoir les classes $\rightarrow$ (toujours suivi de), $\leftarrow$ (toujours précédé de) et $\nrightarrow$ (jamais suivi de). Nous considérons, en plus de la portée « Global », qui était celle considérée par défaut dans le chapitre 3, deux portées nommées « Before » et « After ». La définition d'une portée réduit la fenêtre temporelle dans laquelle l'invariant est évalué. Dans cette proposition, les invariants calculés sont définis par trois éléments : une propriété qui doit être vraie dans la fenêtre définie ; une position qui identifie un moment dans l'exécution où la fenêtre commence ou se termine ; et une direction (Before ou After) qui va préciser l'orientation de la fenêtre par rapport à la position. Plusieurs solutions pourraient être adoptées pour définir une position. Il est possible par exemple de s'appuyer sur des états particuliers du treillis. Dans le cadre de cette thèse, nous avons envisagé de faire référence aux événements minimaux liés à chaque type d'événement. Ainsi, dans le cas où l'ensemble d'événements minimaux associé à un type d'événement c ne contient qu'un événement, nous le notons $min(c)$ et nous parlons de la première occurrence d'un événement de type c. Un invariant que nous souhaitons prendre en compte serait, par exemple, l'invariant « a précède toujours b après la première occurrence de c » et serait noté « a $\leftarrow$ b After min(c) ». Cet invariant serait satisfait par une séquence

d'événements comme <b, c, a, b>. Avec la même séquence mais avec une portée « Global », la propriété a ← b n'est pas vraie. Ces nouvelles définitions de portée permettent donc d'exprimer de nouveaux invariants qui viennent compléter ceux qui ont été définis avec la portée « Global ».

6.1.2.2 Inference des portées

L'inférence d'invariants prenant en compte les portées « Before » et « After » à partir d'une trace d'exécution se fait en trois phases pouvant être réalisées en parallèle avec l'approche exposée dans le chapitre 3. Ces phases reprennent des algorithmes déjà présentés dans le chapitre 3.

Une première phase consiste à calculer toutes les positions qui vont être utilisées. Le calcul effectué dans cette phase reprend ce qui a été fait dans la section 3.2.3.4. Les séries de coordonnées de points du treillis qui correspondent aux différents événements $mins(a)$ sont stockées dans une hashmap nommée *thresholdMap*. A chaque position correspond une fenêtre avant et une fenêtre après pour lesquels il va falloir déterminer quels invariants sont satisfaits.

Algorithm 12 calcul des minmax pour les différents seuils

```
 1: function MINMAXTHRESHOLD
 2:     for transition in transitionlist do
 3:         for event in thresholdMap.keys() do
 4:             if transition.isInferiorTo(thresholdMap.get(event)) then
 5:                 minmaxBeforeMap.get(event).updateMinMax(transition)
 6:             else if transition.isSuperiorTo(thresholdMap.get(event)) then
 7:                 minmaxAfterMap.get(event).updateMinMax(transition)
 8:             end if
 9:         end for
10:     end for
11: end function
```

Une fois l'ensemble des positions calculées, lors d'une deuxième phase, un calcul d'ensemble minimaux et maximaux est refait par rapport à chaque position (afin de tenir compte des deux fenêtres temporelles associées à la position). Pour cela, l'approche présentée dans la section 3.2.3.4 est réutilisée. L'algorithme 12 détaille le fonctionnement de ce calcul. Dans cet algorithme, l'ensemble des transitions du treillis est comparé par rapport aux différentes positions. Si une transition est inférieure à la position calculée pour un type d'événement (ligne 4), les min et max avant cette position seront mis à jour (ligne 5). Une approche similaire est effectuée aux lignes 6 et 7 si la transition est supérieure à la position. Ainsi, pour chaque position de la hashmap *thresholdMap*, on va

calculer deux séries de min et de max : *minmaxBeforeMap* et *minmaxAfterMap* afin de tenir compte de la fenêtre temporelle située avant la position et de la fenêtre temporelle située après la position.

Algorithm 13 calcul des invariants avec les scopes

```
1: function invConstScope
2:     for event in eventlist do
3:         minmaxBeforeMap.get(event).invInference("before", event)
4:         minmaxAfterMap.get(event).invInference("after", event)
5:     end for
6: end function
```

Enfin, l'algorithme 13 décrit la génération d'invariants avec des portées Before et After. Cette génération reprend un algorithme similaire à l'algorithme 5 décrit dans la section 3.2.3.5 en ajoutant à l'invariant la direction ($Before$ ou $After$) et la position $event$.

Les complexités des phases deux et trois de l'approche détaillée dans cette section correspondent à celle exposées dans la section 3.2.3 en ajoutant un facteur multiplicatif de respectivement t et $2t$ pour chaque complexité (t étant le nombre de types d'événements). L'approche demande donc en théorie plus de temps de calcul que l'approche traditionnelle.

6.1.2.3 Fusion et détection

Les processus de fusion et de détection n'ont pas été réalisés dans le cadre de cette thèse. Ces processus sont plus complexes que ceux détaillés dans l'approche présentée dans le chapitre 3 et nécessitent encore un travail de réflexion et de développement. Il est cependant possible d'exposer quelques pistes permettant de réaliser ces deux processus.

En ce qui concerne la fusion, il est possible de reprendre les principes exposés dans la section 3.3.2 et de comparer les invariants qui ont les mêmes couples <position, direction> entre eux. Il est aussi possible de tenir compte de la portée « Global fg. Par exemple, garder l'invariant « a → b after min(c) » n'est peut être pas nécessaire si l'invariant « a → b » est aussi vrai pour la même exécution en considérant la portée « Global ».

L'introduction de nouvelles portées au mécanisme de détection nécessite la gestion de nouveaux automates pour évaluer les traces avec les nouveaux invariants. La vérification peut être réalisée, comme pour les invariants avec une portée globale, pendant l'évaluation de la trace où à la fin de l'évaluation de celle-ci. Pour cela, une création de nouveaux automates comme ceux exposés dans le tableau 3.4 est possible.

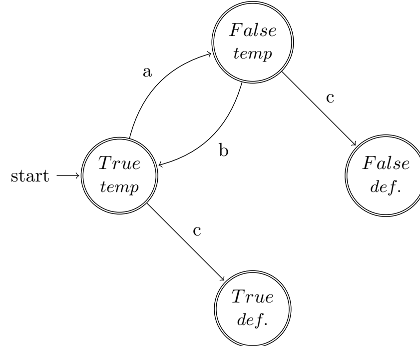

FIGURE 6.2 – Automate correspondant à l'invariant a → b before min(c)

La figure 6.2 présente un automate pouvant être construit pour vérifier l'invariant « a → b before min(c) ». Cet invariant peut être utilisé en cours d'évaluation de la trace à l'aide des états « True def. » et « False def. ».

6.1.2.4 Évaluation

La phase de construction a été implémentée pour la proposition d'invariants comportant des portées. L'évaluation de cette proposition porte donc sur cette étape en particulier. Elle a pour objectif d'avoir une première estimation de l'impact de la proposition sur l'approche en terme de passage à l'échelle. Pour cela, plusieurs modèles intermédiaires correspondant à des exécutions de durées différentes sont construits.

Phase	construction		
Durée d'exécution	2 min	5 min	*ratio*
Taille de la trace (événements)	212	324	1.53
Taille de la liste d'invariants (portée globale)	3118	4622	1.48
Coût en temps (minutes)	13.08	38.17	2.91
Taille de la liste d'invariants (toutes les portées)	140034	155360	1.11

TABLEAU 6.3 – Evaluation de la proposition d'utiliser différentes portées

On retrouve dans le tableau 6.3 les résultats liés à l'utilisation de portées pour les invariants de causalité. On observe que le coût en temps lié à cette proposition est important. Il est de 12 minutes (resp. 38 minutes) pour générer la liste d'invariants pour une exécution de 2 minutes (resp. 5 minutes). Le ratio lié à ce coût est supérieur au ratio lié à la taille de la trace. Il tend à montrer que la solution ne passe pas à l'échelle. Dans ce tableau, on peut aussi remarquer que la taille de la liste d'invariants utilisant différentes portées est plus grande que pour les invariants considérant l'ensemble de l'exécution. La taille de cette liste semble stable si la durée d'exécution augmente (ratio proche de 1).

La taille importante de ce type de modèle intermédiaire peut avoir un impact sur le temps nécessaire à la fusion des différentes listes d'invariants. Il est par ailleurs difficile d'estimer la taille de la liste après une phase de fusion ainsi que l'impact de cette taille sur le processus de détection. Après une implémentation des briques manquantes, il pourrait être intéressant d'évaluer ces différents points. Cependant, les premiers résultats mitigés laissent à penser qu'une énumération exhaustive de tous les invariants n'est peut être pas la solution appropriée.

6.2 N-grammes

Les N-grammes ont été introduits par Shannon en 1951 [Sha51]. Un N-gramme correspond à une sous-séquence de n éléments construite à partir d'une séquence donnée. Ils peuvent être associés à un modèle probabiliste qui correspond à la probabilité d'apparition des différents N-grammes. Ces probabilités permettent d'obtenir une vraisemblance d'apparition pour chaque N-gramme. Ils peuvent être utilisés en traitement de langages ou en traitement du signal.

Par exemple, pour une séquence <a, b, a, c>, on obtient les 2-grammes suivants : « ab », « ba » et « ac ». On peut aussi calculer les probabilités d'apparition des différents éléments de la séquence : $P(a|b) = 1$, $P(b|a) = 1/2$ et $P(c|a) = 1/2$.

6.2.1 Proposition

Notre proposition consiste à générer un automate à partir de N-grammes. L'automate construit à l'aide de ce procédé sera considéré comme une alternative à l'automate généralisé. Pour construire l'automate, une liste de N-grammes sera générée pour chaque trace d'exécution correcte. Ces listes seront ensuite fusionnées entre elles pour obtenir une liste de N-grammes représentant l'ensemble des exécutions. Enfin, les N-grammes seront utilisés pour construire un automate représentant un modèle de comportement normal de l'application. L'approche proposée, tout comme l'approche utilisant l'algorithme de généralisation k-tail, repose sur des liens de causalité directs pour générer un automate. Le processus de construction et l'automate résultant sont cependant différents. La construction d'un automate à l'aide de N-grammes n'a lieu que lors d'une phase de fusion. Cette phase considère l'ensemble des exécutions utilisées pour construire un automate généralisé. Avec cette proposition, on essaye de construire une automate généralisé sans passer par une phase de généralisation comme dans k-tail. Les probabilités d'occurrence des N-grammes ne sont pas considérées dans l'automate inféré.

6.2.2 Inférence de N-grammes

L'inférence d'une liste de N-grammes intervient lors de la construction du treillis pour une exécution. Elle se fait pour toutes les exécutions utilisées pour construire le modèle. Pour cela, une modification de l'algorithme 1 est nécessaire. La fonction LatticeConstruct(state, trace, hist, ngram) comprend deux nouveaux paramètres : *hist* et *ngram*. Le paramètre *hist* correspond à l'historique de la séquence d'événements utilisé pour construire le N-gramme. Le paramètre *ngram* correspond à la longueur du N-gramme.

Algorithm 14 Inférence de N-grammes

```
 1: function LATTICECONSTRUCT(state, trace, hist, ngram)
 2:     for i in Dimention do
 3:         event ← trace.getEvent(state, i)
 4:         if canEvolve(state, event) then
 5:             newstate ← State(state, i, event)
 6:             newtrans ← Transition(state, newstate, event)
 7:             newhist ← hist
 8:             updateMinMax(newtrans)
 9:             transitionset.add(newtrans)
10:             if ngram > 1 then
11:                 if hist.size() == ngram − 1 then
12:                     ngramtemp ← newNgram(hist, event)
13:                     ngramlist.add(ngram)
14:                     newhist ← newList()
15:                     for i ← 1 to hist.size() do
16:                         newhist.add(hist.get(i))
17:                     end for
18:                 end if
19:                 newhist.add(event)
20:             end if
21:             if stateset.add(newstate) then
22:                 LATTICECONSTRUCT(newstate, trace, newhist, ngram)
23:             else if ngram > 1 then
24:                 EXPLORE(newstate, trace, newhist, ngram, ngram − 1)
25:             end if
26:         end if
27:     end for
28: end function
```

L'algorithme 14 détaille un version modifiée de l'algorithme 1. L'algorithme comporte des ajouts qui correspondent à la génération de N-grammes. Aux lignes 10 et 23 un test est effectué pour voir si la construction de N-grammes est possible. Si l'historique

d'événements contient suffisamment d'éléments pour construire un N-gramme, on le construit et on l'ajoute à la liste des N-grammes (lignes 11 à 13). Un nouvel historique est calculé pour les parcours ultérieurs (lignes 14 à 17) et l'événement courant est ajouté au nouvel historique. Enfin, un appel récursif est effectué à ligne 22 ou à la ligne 24. Le second appel récursif sert à explorer les états du treillis déjà construits pour générer les N-grammes existants à profondeur $ngram - 1$.

6.2.3 Fusion de N-grammes et construction de l'automate

La fusion des listes de N-grammes des différentes exécutions s'effectue en deux parties. La première partie consiste à fusionner les listes de N-grammes de plusieurs exécutions. Ce processus de fusion conserve l'ensemble des N-grammes tout en supprimant les doublons. La liste de N-grammes obtenue à la suite du processus de fusion va être utilisée lors d'une seconde étape pour construire un automate.

L'algorithme 15 présente une proposition de construction d'automate à partir d'une liste de N-grammes. Dans cet algorithme, un parcours de la liste de N-grammes (ligne 3 à 27) va pour chaque N-gramme réaliser deux étapes. La première consiste à mettre en place deux structures Hashmap *premap* (lignes 3 à 8) et *postmap* (ligne 9 à 14) qui sont liées respectivement au N-1 préfixes et suffixes des N-grammes. Ces deux structures sont utilisées lors de la seconde étape (ligne 15 à 26) pour la construction de l'automate en faisant le lien entre les différents N-grammes. Si deux N-grammes partagent leurs préfixes et suffixe une transition est crée. La transition a pour structure $<ngram1, ngram2, event>$. Cette structure permet de faire le lien entre les deux N-grammes. Pendant la création des transitions tous les $ngram2$ sont marqués. Les N-grammes non marqués correspondent à des N-grammes dont les préfixes ne correspondent pas aux suffixes des autres N-grammes. Il est donc nécessaire de décomposer les N-grammes non marqués en plusieurs états d'automate dans une étape ultérieure pour obtenir un automate complet.

La construction d'un automate à partir d'une liste de N-grammes a une complexité temporelle en en $\mathcal{O}(g \times \log(g))$ ou g correspond au nombre de N-grammes présents dans la liste.

Algorithm 15 Construction de l'automate à l'aide des n-grammes

1: **function** NGRAMMERGER(ngramlist)
2:     **for** *ngram* in *ngramlist* **do**
3:         $prelist \leftarrow premap.get(ngram.getPrelist())$
4:         **if** $prelist == null$ **then**
5:             $premap.put(ngram.getPrelist(), newList().add(ngram)$
6:         **else**
7:             $prelist.add(ngram)$
8:         **end if**
9:         $postlist \leftarrow postmap.get(ngram.getPostlist())$
10:        **if** $postlist == null$ **then**
11:            $postmap.put(ngram.getPoslist(), newList().add(ngram)$
12:        **else**
13:            $postlist.add(ngram)$
14:        **end if**
15:        $past \leftarrow premap.get(ngram.getPostlist())$
16:        **if** $past \neq null$ **then**
17:            **for** $ngram2$ in $past$ **do**
18:                $transitionset.add(newTrans(ngram, ngram2, ngram2.getLast()))$
19:            **end for**
20:        **end if**
21:        $futur \leftarrow postmap.get(ngram.getPrelist())$
22:        **if** $futur \neq null$ **then**
23:            **for** $ngram2$ in $futur$ **do**
24:                $transitionset.add(newTrans(ngram2, ngram, ngram.getLast()))$
25:            **end for**
26:        **end if**
27:    **end for**
28: **end function**

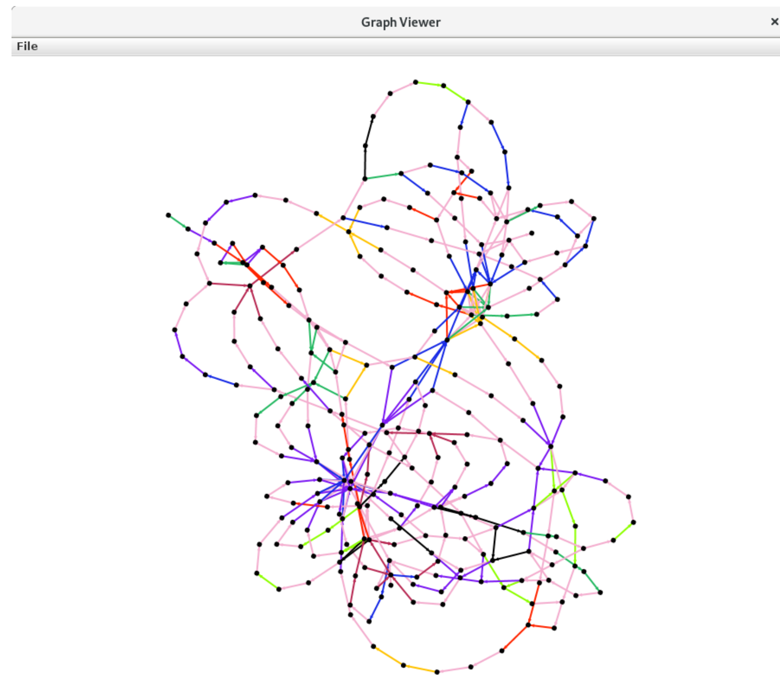

FIGURE 6.3 – Automate construit à l'aide de N-grammes pour une exécution

Dans la figure 6.3, Un automate correspondant à une trace d'exécution correcte du système de fichiers XtreemFS est représenté à l'aide d'un outil de visualisation développé dans le cadre de cette thèse. Chaque point de cet automate correspond à un état et chaque transition à un type d'événement. Les couleurs sont attribuées en fonction du type d'événement.

6.2.4 Différences avec l'automate généralisé et détection

L'automate généré à l'aide de N-grammes présente des caractéristiques similaires à un automate généré à l'aide d'un algorithme de généralisation comme k-tail. Dans les deux cas, les automates acceptent les traces apprises et de nouveaux comportements (notamment grâce aux boucles introduites).

Cependant les deux approches présentent des différences. Certaines différences n'impactent pas le fonctionnement du mécanisme de détection comme le fait que l'automate généré à l'aide de N-grammes est déterministe. D'autres ont un impact sur ce mécanisme et nécessitent une modification d'algorithmes. Une première différence entre les deux approches concerne la présence de comportements distincts au sein d'une même application. Si les listes de N-grammes générés à l'aide de traces ne présentent aucun N-gramme communs, plusieurs automates peuvent être générés. Une seconde différence est

que plusieurs N-grammes peuvent être considérés comme des N-grammes initiaux. Autrement dit, plusieurs N-grammes peuvent correspondre aux débuts des traces utilisées lors de l'apprentissage.

Pour les différences qui présentent un impact sur le mécanisme de détection, il est possible de conserver l'ensemble des N-grammes initiaux pour soit fusionner leurs états initiaux soit considérer plusieurs états initiaux pour la phase de détection.

6.2.5 Évaluation

L'approche vise à proposer une alternative à la génération d'automates généralisés. Pour l'évaluer, une comparaison entre la génération d'automates à l'aide de N-grammes et la génération d'un automate généralisé à l'aide de l'algorithme k-tail est donc nécessaire. Cette comparaison est réalisée en terme de passage à l'échelle (la partie détection n'étant pas implémentée) à l'aide de trois expérimentations.

La première expérimentation vise à évaluer la phase de construction de l'automate intermédiaire à l'aide de traces d'exécution en terme de consommation mémoire et de temps de construction. Dans cette expérimentation, l'approche de construction de N-grammes est comparée à la proposition de génération d'automate à l'aide de n chemins. Pour les comparer, une moyenne est effectuée entre les valeurs obtenues à partir de traces de 2 minutes et de traces de 5 heures.

Phase		construction		
Durée d'exécution		2 min	5 hr	*ratio*
Temps de construction (secondes)	Aut-n	5.046	178	35.6
	Aut-n + 2-gram	5.621	200	35.7
Nombre de transitions	Aut-n	2502	87266	34.9
Nombre de n-grammes	2-gram	267	674	2.5

TABLEAU 6.4 – Evaluation des performances d'un modèle fondé sur les N-grammes

Le tableau 6.4 présente une comparaison entre les approches `Aut-n` et `2-gram`. On observe dans ce tableau un overhead faible pour le temps de construction du modèle intermédiaire. Cet overhead est lié au temps nécessaire pour générer les 2-grammes. Le ratio de cet overhead semble stable si on considère des exécutions de 2 minutes ou des exécutions de 5 heures. En ce qui concerne la taille, on observe que le nombre de 2-grammes est inférieur au nombre de transitions et que le ratio associé est aussi inférieur. La génération de 2-grammes ne semble pas très coûteuse en temps et en espace. L'évolution en taille semble être moins importante que pour un automate ce qui pourrait avoir un impact sur le processus de génération d'automates.

Dans la seconde expérimentation, on cherche à mesurer l'impact du nombre de traces sur la construction d'un automate à l'aide de n-grammes. Ainsi, plusieurs modèles sont construits à l'aide d'un nombre variable de traces. Une comparaison est effectuée entre l'automate généré à l'aide de n-grammes (2-grammes dans le cas de l'expérimentation) et d'automates généralisés avec k-tail selon les stratégies `Mod-n-R-1` et `Mod-n-S-1`.

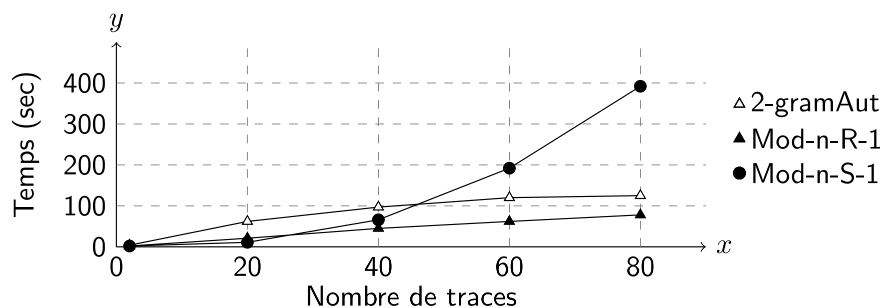

FIGURE 6.4 – Comparaison temporelle des fusions d'automates

La figure 6.4 présente une comparaison des temps de plusieurs approches. On observe dans cette figure que la génération d'automates à l'aide de 2-grammes est globalement plus couteuse en temps que l'approche `Mod-n-R-1`. Cependant, plus le nombre de traces utilisées pour construire un modèle de comportement est important moins cette différence entre les deux approches est importante. On observe aussi qu'elle nécessite un temps de calcul moins important que l'approche `Mod-n-S-1` pour un nombre de trace supérieur à 50. L'approche semble donc passer à l'échelle en terme de temps de construction du modèle.

La troisième et dernière expérimentation vise à étudier les différences de tailles entre les automates générés à l'aide de différentes stratégies. Pour cela, des modèles sont construits selon les stratégies `2-gramAut`, `Mod-n-R-1` et `Mod-n-R-1` pour un nombre variable de traces.

Phase		fusion et généralisation			
Nombre de traces		20	40	60	80
	Mod-n-S-1	228	187	179	173
Nombre de transitions	Mod-n-R-1	115	74	82	87
	2-gramAut	29871	37082	39142	40528

TABLEAU 6.5 – Comparaison quantitative des fusion d'automates

Dans le tableau 6.5 on observe les différences en taille entre les différentes approches. On observe que les automates générés à l'aide d'un processus de généralisation (`Mod-n-R-1` et `Mod-n-R-1`) sont dans ordre de grandeur similaire même si une différence

est notable. Pour l'automate généré à partir de 2-grammes, on observe en revanche que le nombre de transitions est bien plus important. Une autre différence se situe dans le nombre croissant de transitions pour l'approche utilisant des n-grammes alors que ce nombre tend à décroître pour les approches utilisant un processus de généralisation. La différence entre les deux approches s'explique par le fait que l'algorithme de généralisation a tendance à sur-généraliser ce qui donne un automate composé d'un faible nombre d'états. En revanche, l'algorithme de génération d'automates à l'aide de n-grammes cherche à construire un automate déterministe à l'aide des n-grammes.

6.3 Conclusion

Ce chapitre propose trois types de modèles de comportements alternatifs ou additionnels aux modèles présents dans l'approche détaillée dans le chapitre 3 ainsi qu'une évaluation partielle de ces modèles.

Le premier modèle proposé est une liste d'invariants d'occurrence. La proposition détaille le fonctionnement, les étapes nécessaires pour générer une liste à l'aide de traces et l'utilisation de la liste d'invariants d'occurrence pour la phase de détection. Une évaluation de la proposition a ensuite été effectuée. Elle a permis de vérifier que l'approche passait à l'échelle. Les résultats de détection sont intéressants puisque le modèle proposé est complémentaire aux modèles existants.

Une deuxième proposition a consisté à introduire les notions de « scopes » (présentés dans l'article [DAC99]) pour les invariants de causalité ainsi que la méthode pour les générer. Les premières évaluations qui ont porté sur la phase de génération d'une liste correspondant à une trace d'exécution ont montré que la solution semblait passer à l'échelle en taille (malgré un nombre important d'invariants) mais pas en temps de génération. Cette proposition nécessite actuellement des améliorations pour passer à l'échelle et l'implémentation des étapes de fusion et de détection est à réaliser.

La troisième et dernière proposition visait à construire un automate alternatif sans passer par une phase de « fusion & généralisation ». Elle consistait à construire une liste de N-grammes dans l'optique de générer un automate. L'évaluation de cette proposition a concerné le passage à l'échelle. Les automates générés par cette méthode passent à l'échelle. Une modification de l'approche est cependant nécessaire pour la phase de détection. Elle devra notamment mettre en évidence les états initiaux de l'automate pour permettre son exploration.

Chapitre 7

Conclusion et travaux futurs

Nous nous proposons de récapituler les contributions apportées par les travaux réalisés pendant cette thèse puis de présenter quelques perspectives issues de problématiques soulevées lors de la réalisation de ces travaux.

7.1 Contributions

Cette thèse s'est positionnée sur la problématique de détection d'intrusion dans les applications distribuées et plus particulièrement sur la proposition d'une approche comportementale comme alternative à la corrélation d'alertes. L'étude de l'état de l'art a permis de situer nos travaux par rapport aux travaux déjà publiés dans trois domaines : la détection d'intrusion, la représentation des systèmes distribués et l'apprentissage de modèles de comportement d'applications. Cette étude nous a amené à constater qu'il est possible de construire un modèle de comportement d'une application à l'aide de traces d'exécutions. À notre connaissance, ce type d'approche a été peu utilisé pour détecter des attaques : la plupart des travaux concernent plutôt la mise au point des applications (détection d'erreurs). Les principaux problèmes identifiés sont en lien avec la phase d'apprentissage et concernent le passage à l'échelle des approches proposées. Les constats effectués nous ont permis d'orienter nos travaux de recherche vers la proposition d'une approche de modélisation d'une application distribuée pour la détection d'intrusion. Les contributions principales de cette thèse sont au nombre de quatre et sont présentées dans les paragraphes suivants.

La première contribution consiste en une proposition d'approche de détection d'intrusion basée sur la modélisation du comportement d'une application distribuée. Cette approche est composée de deux phases. La première phase a pour objectif d'inférer le modèle de comportement d'une application distribuée à l'aide de traces d'exécution. La seconde phase vise à évaluer une trace à l'aide du modèle de comportement inféré pour

déterminer si la trace correspond à une exécution correcte ou incorrecte. Plusieurs types de modèles ont été exposés lors de la présentation de cette approche. Les méthodes d'inférence et de détection de ces différents types de modèle ont été détaillées. A ce stade, nos propositions sont fortement connectées à des travaux existants. Néanmoins, un point original de notre approche réside dans le fait de considérer un modèle composé de plusieurs sous-modèles distincts tout en cherchant à combiner et factoriser les étapes de leurs calculs. Dans le cas du modèle à base d'invariants, le fait que la définition des classes d'invariants, le processus de calcul de la liste d'invariants et les mécanismes de réduction de cette liste s'appuient sur les même ensembles d'événements (minimaux et maximaux) est aussi une démarche nouvelle.

La deuxième contribution est basée sur l'étude des problèmes de passage à l'échelle. Les problèmes exposés sont soit connus et liés aux systèmes distribués (croissance exponentielle du treillis) soit liés à l'approche (généralisation d'automates de grande taille). À la suite de cette étude, nous avons présenté deux propositions pour passer à l'échelle. La première proposition a pour objectif de traiter le problème de la croissance exponentielle du treillis. Elle consiste à explorer un nombre fini de chemins d'exécution pour obtenir une structure qui correspond à un sous ensemble du treillis. La seconde proposition cherche à résoudre le problème de la généralisation d'automates de grandes taille à l'aide d'algorithmes comme k-tail. Une fusion et généralisation itérative est ainsi proposée. Les deux propositions ont pour objectif de résoudre les problèmes de passage à l'échelle sans modifier le principe général des méthodes de génération et en limitant l'impact sur les résultats de détection.

La troisième contribution concerne l'évaluation de l'approche de détection d'intrusion et des propositions pour passer à l'échelle. Cette évaluation prend en compte la description de deux cas d'étude et de différents jeux de données. Elle a permis de confirmer les problèmes de passage à l'échelle et de valider l'utilité des deux propositions. On a pu noter aussi dans cette évaluation que l'approche permettait effectivement de détecter les attaques réalisées contre une application réelle et que les propositions avait un impact faible sur les résultats de détection. Cette similitude dans les résultats de détection est due en partie à la complémentarité des différents types de modèles. L'évaluation a aussi permis de montrer qu'il était possible d'utiliser un sous-ensemble de traces ainsi que des traces d'exécutions courtes pour générer un modèle qui accepte plus de traces ainsi que des traces d'exécution longues.

Enfin, la quatrième et dernière contribution a été de proposer d'autres types de modèles complémentaires ou alternatifs. Le premier type de modèle proposé se base sur le nombre d'occurrence des différents types d'évènements. Le second type de modèle introduit une notion de portée pour les invariants de causalité. Le troisième et dernier

types de modèle correspond à un automate généré à l'aide de n-grammes. Les différents types de modèles ont été complètement évalués pour le premier et partiellement évalués pour les deux autres. L'évaluation de ces types de modèles confirme l'intérêt d'utiliser plusieurs types de modèles complémentaires, souligne des éventuels problèmes de passage à l'échelle et expose des difficultés d'adaptation aux algorithmes existants.

7.2 Perspectives

Les perspectives d'évolutions ou d'approfondissements des travaux réalisés dans cette thèse peuvent se diviser en plusieurs axes qui sont la proposition de nouveaux types de modèles, la variation de la granularité des données, la combinaison des modèles et l'enrichissement des jeux de données.

Proposition de nouveaux types de modèles : Dans cette thèse, nous avons proposé plusieurs types de modèles. Parmi ceux-ci, trois ont été entièrement implémentés et évalués (automates généralisés, invariants de causalité et invariants d'occurrence) et deux n'ont pas été entièrement implémentés. Il s'agirait donc dans un premier temps de continuer sur les propositions figurant dans le chapitre 6. En ce qui concerne la proposition de scopes pour les invariants, il faudrait définir une logique de fusion qui prend à la fois en compte les invariants utilisant un scope global et non global, préciser les automates utilisés pour vérifier les invariants lors de la détection, intégrer les automates à l'algorithme de détection et évaluer le sur-coût lié à l'introduction de ce type de modèle dans le processus de détection (nombre important d'invariants). Pour la proposition concernant les n-grammes, il serait nécessaire d'améliorer l'approche en introduisant les notions de points d'entrée pour l'exploration de l'automate lors de la phase de détection, de comparer les résultats de détection entre l'automate généré à l'aide de n-grammes et l'automate généralisé à l'aide de k-tail, de comparer avec une approche où on utiliserait directement les n-grammes et non l'automate. Dans un second temps, des propositions de nouveaux types de modèles pourraient être effectuées. Il serait par exemple possible d'introduire de nouveaux modèles basés sur d'autres caractéristiques présentes dans les applications ou d'utiliser des modèles comme les réseaux de neurones sur les logs des applications. Ces propositions de nouveaux types de modèles pourraient être accompagnées de propositions pour passer à l'échelle.

Variation de la granularité des données : La granularité des données est en lien avec la précision des informations monitorées. Dans les travaux effectués dans le cadre de cette thèse, cette problématique n'a pas été explorée. On ne considère que l'événement

(généré par l'application après instrumentation) et une définition possible de type d'événement. Il pourrait être intéressant de faire varier cette granularité. On pourrait par exemple considérer les regroupements de certains couples ou ensemble d'événements. On pourrait aussi proposer d'autres définitions des types d'événements (envoi de message, messages concernant certains canaux, ...) ou encore de collecter des informations à des niveaux différents (ptrace).

Combinaison des modèles : La combinaison des modèles intervient dans la détection. Il peut être intéressant d'étudier les différentes façon d'utiliser les modèles ensemble. Actuellement, nous utilisons trois types de modèles au sein d'un même algorithme de détection (automate, invariants de causalité et invariants de communication) et un autre type de modèle (les invariants d'occurrence) est utilisé dans un algorithme de détection indépendant. La combinaison de nouveaux types de modèles au sein d'un même algorithme peut poser des problèmes en cas d'ajout de nouveaux modèles (complexité de l'algorithme, modularité). Il pourrait donc être intéressant de proposer une approche de détection modulaire. D'autres part, certains modèles pourraient être redondants. Il serait alors envisageable de considérer une approche pour réduire cette redondance ou au contraire en tirer profits.

Enrichissement des jeux de données : L'évaluation de la proposition effectuée dans le cadre de cette thèse a porté sur deux applications distribuées au travers de traces d'exécutions légitimes et illégitimes (attaques). En ce qui concerne les attaques, le jeu de donnée est actuellement composé de traces concernant 5 attaques dans 4 contextes d'attaques différents. Il serait intéressant de proposer de nouveaux contextes d'attaques ainsi que de nouveaux types d'attaques. L'objectif de cet enrichissement du jeu de donnée serait d'évaluer les capacités de détection des modèles existants et des nouveaux modèles proposés. La collecte de nouvelles traces légitimes semble aussi intéressante. Cela permettrait par exemple, dans le cas de l'application de commerce en ligne, d'effectuer une évaluation plus poussée sur la complétude des modèles. Il semble donc important d'enrichir les jeux de données.

Annexe A

Liste des publications

David Lanoe, Eric Totel et Michel Hurfin. Apprentissage d'un modèle de comportement d'une application distribuée pour la détection d'intrusion. Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI). Mai 2018.

David Lanoe, Eric Totel et Michel Hurfin. A scalable and efficient correlation engine to detect multistep attacks in distributed systems. Symposium on Reliable Distributed Systems (SRDS). Octobre 2018.

David Lanoe, Eric Totel, Michel Hurfin et Carlos Maziero. Modélisation du comportement hybride d'une application distribuée pour la détection d'intrusion. Rendez-Vous de la Recherche et de l'Enseignement de la Sécurité des Systèmes d'Information (RESSI). Mai 2019.

David Lanoe, Eric Totel, Michel Hurfin et Carlos Maziero. An Efficient and Scalable Intrusion Detection System on Logs of Distributed Applications. IFIP International Conference on ICT Systems Security and Privacy Protection (IFIPSEC). Juin 2019.

Bibliographie

[ABL02] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1) :4–16, 2002.

[AC+10] Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4 :40–79, 2010.

[ANS04] Agence nationale de la sécurité des systèmes d'information, ANSSI. Guide d'élaboration de politiques de sécurité des systèmes d'information. https://www.ssi.gouv.fr/guide/pssi-guide-delaboration-de-politiques-de-securite-des-systemes-dinformation/, 2004. (Dernière visite le 27 Juillet 2019).

[BABE11] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D Ernst. Synoptic : Studying logged behavior with inferred models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 448–451. ACM, 2011.

[BB75] Lenore Blum and Manuel Blum. Toward a mathematical theory of inductive inference. *Information and control*, 28(2) :125–155, 1975.

[BBEK14] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

[BF72] Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6) :592–597, 1972.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2) :323–342, 1983.

[CCF+07] Eugenio Cesario, Toni Cortes, Erich Focht, Matthias Hess, Felix Hupfeld, Björn Kolbeck, Jesús Malo, Jonathan Martí, and Jan Stender. The xtreemfs architecture. *Linux Tag*, 2007.

[CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[CW98a] Jonathan E Cook and Alexander L Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7 :215–249, 1998.

[CW98b] Jonathan E Cook and Alexander L Wolf. Event-based detection of concurrency. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 35–45. ACM, 1998.

[DAC99] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002)*, pages 411–420. IEEE, 1999.

[DAH01] Luca De Alfaro and Thomas A Henzinger. Interface automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8) :805–822, 1999.

[Den87] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, pages 222–232, 1987.

[Dev00] Srikrishna Devabhaktuni. *Softspec : Software-based speculative parallelism via stride prediction.* PhD thesis, Master's thesis, MIT/LCS, 2000.

[Dil90] RP Dilworth. Proof of a conjecture on finite modular lattices. In *The Dilworth Theorems*, pages 219–224. Springer, 1990.

[DlH10] Colin De la Higuera. *Grammatical inference : learning automata and grammars.* Cambridge University Press, 2010.

[DM94] Sreerupa Das and Michael C Mozer. A unified gradient-descent/clustering architecture for finite state machine induction. In *Advances in neural information processing systems*, pages 19–26, 1994.

[ECGN01] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2) :99–123, 2001.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1) :110–121, 2005.

[Fid87] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.

[FM82] Michael J Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 70–75. ACM, 1982.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software : Practice and Experience*, 16(3) :225–233, 1986.

[Gar12a] Vijay K Garg. Lattice completion algorithms for distributed computations. In *International Conference On Principles Of Distributed Systems*, pages 166–180. Springer, 2012.

[Gar12b] Vijay K Garg. *Principles of distributed systems*, volume 3144. Springer Science & Business Media, 2012.

[GCB18] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1149–1159. ACM, 2018.

[Gol67] E Mark Gold. Language identification in the limit. *Information and control*, 10(5) :447–474, 1967.

[GTM09] Laurent George, Valérie Viet Triem Tong, and Ludovic Mé. Blare tools : A policy-based intrusion detection system automatically set by the security policy. In *International Workshop on Recent Advances in Intrusion Detection*, pages 355–356. Springer, 2009.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet : proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[Kom43] Atuo Komatu. 27. on a characterisation of order-preserving mapping-lattice. *Proceedings of the Imperial Academy*, 19(3) :125–128, 1943.

[KRL97] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems : A specification-based approach. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 175–187. IEEE, 1997.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[LFY+10] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Jiang Li, and Bin Wu. Mining program workflow from interleaved traces. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 613–622. ACM, 2010.

[LHT18] David Lanoe, Michel Hurfin, and Eric Totel. A scalable and efficient correlation engine to detect multi-step attacks in distributed systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 31–40. IEEE, 2018.

[LJ88] Teresa F Lunt and R Jagannathan. A prototype real-time intrusion-detection expert system. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*, pages 59–66. IEEE, 1988.

[LK06] David Lo and Siau-Cheng Khoo. Quark : Empirical assessment of automaton-based specification miners. In *2006 13th Working Conference on Reverse Engineering*, pages 51–60. IEEE, 2006.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510. ACM, 2008.

[LMS12] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended fsa from software : An empirical assessment. *Journal of Systems and Software*, 85(9) :2063–2076, 2012.

[LMTY02] Leslie Lamport, John Matthews, Mark R Tuttle, and Yuan Yu. Specifying and verifying systems with tla+. In *ACM SIGOPS European Workshop*, pages 45–48, 2002.

[Lyn88] Nancy Lynch. I/o automata : A model for discrete event systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1988.

[M+88] Friedemann Mattern et al. *Virtual time and global states of distributed systems.* Citeseer, 1988.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models : Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4) :364–380, 2001.

[MQ88] L Miclet and J Quinqueton. Syntactic and structural pattern recognition, volume 45 of nato asi series f : Computer and systems sciences, 1988.

[Pax99] Vern Paxson. Bro : a system for detecting network intruders in real-time. *Computer networks*, 31(23-24) :2435–2463, 1999.

[Pfa06] John L Pfaltz. Using concept lattices to uncover causal dependencies in software. In *Formal Concept Analysis*, pages 233–247. Springer, 2006.

[PMM17] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail : Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–411. IEEE, 2017.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems : a survey of current trends. In *Current trends in Concurrency*, pages 510–584. Springer, 1986.

[PW93] Leonard Pitt and Manfred K Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)*, 40(1) :95–142, 1993.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering—Esec/Fse'97*, pages 432–449. Springer, 1997.

[SBC+10] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D Ernst, and Yuriy Brun. Synoptic : Summarizing system logs with refinement. In *SLAML*, 2010.

[SBDB00] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 144–155. IEEE, 2000.

[SBN+16] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 291–309, 2016.

[Sha51] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1) :50–64, 1951.

[Sma88] Stephen E Smaha. Haystack : An intrusion detection system. In *[Proceedings 1988] Fourth Aerospace Computer Security Applications*, pages 37–44. IEEE, 1988.

[THH+16] Eric Totel, Mouna Hkimi, Michel Hurfin, Mourad Leslous, and Yvan Labiche. Inferring a distributed application behavior model for anomaly based intrusion detection. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 53–64. IEEE, 2016.

[TJC+98] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244)*, pages 260–267. IEEE, 1998.

[TVM04] Eric Totel, Bernard Vivinis, and Ludovic Mé. A language driven intrusion detection system for event and alert correlation. In *IFIP International Information Security Conference*, pages 209–224. Springer, 2004.

[Val96] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[Val06] Fredrik Valeur. *Real-time intrusion detection alert correlation.* University of California, Santa Barbara, 2006.

[VVKK04] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A Kemmerer. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing*, 1(3) :146–169, 2004.

[War38] Morgan Ward. Structure residuation. *Annals of Mathematics*, pages 558–568, 1938.

[YACK11] Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostic. Finding almost-invariants in distributed systems. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 177–182. IEEE, 2011.

[YEB+06] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta : mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pages 282–291. ACM, 2006.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

Titre : Construction d'un multi-modèle d'application répartie pour la détection d'intrusion

Mot clés : détection d'intrusion, apprentissage automatique, systèmes distribués

Résumé : L'étude réalisée durant cette thèse porte sur la sécurité des applications distribuées. Bien que les problèmes de sécurité soient traités lors du développement des applications, une attaque peut affecter les services fournis ou permettre l'accès à des données confidentielles. Pour détecter les intrusions, nous considérons un mécanisme de détection d'anomalies qui repose sur un modèle du comportement normal de l'application surveillée. Au cours d'une phase de construction du modèle, l'application est exécutée plusieurs fois pour observer certains de ses comportements corrects. Chaque trace collectée permet d'identifier des événements et leurs relations de causalité, sans qu'une horloge globale soit nécessaire. Le modèle construit est double : il comporte un automate et une liste d'invariants probables qui caractérisent tous les deux des séquences d'événements autorisées. Ces deux modèles sont au départ redondants mais cette redondance diminue lorsque des techniques de généralisation sont appliquées à l'automate. Les solutions existantes souffrent de problèmes de passage à l'échelle. Dans cette thèse, des propositions sont faites pour résoudre ces problèmes de passage à l'échelle, tout en conservant une bonne précision pendant la phase de détection. Pour évaluer les solutions proposées, une application distribuée réelle est utilisée et plusieurs attaques contre elle sont envisagées.

Title: Construction of a distributed application multi-model for intrusion detection

Keywords: intrusion detection, machine learning, distributed systems

Abstract: The study carried out during this thesis focuses on the security of distributed applications. Although security concerns are addressed during application development, an attack may affect the services provided or allow access to confidential data. To detect intrusions, we consider an anomaly detection mechanism that is based on a model of the normal behavior of the monitored application. During a model construction phase, the application is run multiple times to observe some of its correct behaviors. Each trace collected makes it possible to identify events and their causal relationships, without the need of a global clock. The model constructed is twofold: it includes an automaton and a list of probable invariants which both characterize authorized sequences of events. These two models are initially redundant but this redundancy decreases when generalization techniques are applied to the automaton. Existing solutions suffer from scalability issues. In this thesis, proposals are made to solve these scalability issues, while maintaining good precision during the detection phase. To evaluate the proposed solutions, a real distributed application is used and several attacks against it are considered.